(19) KOREAN INTELLECTUAL PROPERTY OFFICE

KOREAN PATENT ABSTRACTS

(11)Publication number: 000048532 A
(43)Date of publication of application: 25.07.2000

(21)Application number: 997002442
(22)Date of filing: 22.03.1999
(30)Priority: 23.09.1996 GB 96 9619823

(51)Int. Cl G06F 9/34

(71)Applicant: ARM LIMITED
(72)Inventor: YORK, RICHARD
FRANCES, HEDLEY, JAMES
SYMES, DOMINIC
BILES, STUART

(54) REGISTER ADDRESSING IN A DATA PROCESSING APPARATUS

(57) Abstract:

PURPOSE: Register addressing in a data processing apparatus is provided to only execute once a remapping instruction used to configure a register remapping logic prior to the a repeat instruction being executed.

CONSTITUTION: A data processing apparatus comprises: a plurality of registers for storing data items to be processed; a processor for processing instructions to be applied to data items stored in plurality of registers; and register remapping logic for converting a logical register reference within a preselected set of instructions to a physical register reference identifying the register containing the data item required for processing by the processor. A remapping instruction need only be executed once in order for the remapping to be applied to a desired number of instructions. This is in contrast to prior art techniques, where subsequent to a remapping instruction being executed, the remapping is applied to all subsequent instructions, i.e. a desired number of instructions cannot be selected. The Register addressing in a data processing apparatus is particularly advantageously employed in apparatus arranged to repeat an instruction loop, the instruction loop including said preselected set of instructions. In such cases, loop hardware used to manage the repeat instruction can be arranged to update the register remapping logic each time the instruction loop is repeated, and hence the remapping instruction used to configure the register remapping logic is only executed once prior to the repeat instruction being executed.

Legal Status
Date of request for an examination (20020711)

Final disposal of an application (application)

(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(51) Int. Cl.[6]
G06F 9/34

(11) 공개번호 특2000-0048532
(43) 공개일자 2000년07월25일

(21) 출원번호 10-1999-7002442
(22) 출원일자 1999년03월22일
번역문제출일자 1999년03월22일
(86) 국제출원번호 PCT/GB1997/02258 (87) 국제공개번호 WO 1998/12625
(86) 국제출원출원일자 1997년08월22일 (87) 국제공개일자 1998년03월26일
(81) 지정국 EP 유럽특허 : 오스트리아 벨기에 스위스 독일 덴마크 스페인 프랑스 영국 그리스 아일랜드 이탈리아 룩셈부르크 모나코 네덜란드 포르투칼 스웨덴 핀랜드

국내특허 : 중국 이스라엘 일본 대한민국 러시아

(30) 우선권주장 9619823.9 1996년09월23일 영국(GB)
(71) 출원인 에이알엠 리미티드 맥케이 데이비드 니겔

영국 캠브리지 씨비1 4제이엔 체리힌톤 풀번 로드

(72) 발명자 요크리챠드

영국캠브리지씨비14와이더블유체리힌톤바이올렛클로스6

프렌시스헤들리제임스

영국캠브리지씨비15엔에스웨스톤콜빌웨스톤그린스프링카티지

사이메스도미닉

영국캠브리지씨비14에이치와이체리힌톤웨즈우드드라이브7

빌레스스튜어트

영국캠브리지씨비58엘피뉴마켓로드622

(74) 대리인 이화익

심사청구 : 없음

(54) 데이터 처리장치에서의 레지스터 어드레싱

요약

본 발명은 처리되어야 할 데이터 항목을 저장하는 복수개의 레지스터와, 상기 복수개의 레지스터 내에 저장된 데이터 항목에 인가되어야 할 명령을 처리하는 프로세서와, 사전에 선택된 명령 세트 내의 논리 레지스터 레퍼런스를 프로세서에 의해 처리되어야 할 데이터 항목을 포함하는 레지스터를 식별하는 물리 레지스터 레퍼런스로 변환하는 레지스터 리맵핑 논리를 구비하는 데이터 처리장치를 제공한다. 따라서, 리맵핑 명령은 리맵핑이 소망했던 다수의 명령에 적용되는 순서대로 오직 한 번만 실행되어야 한다. 이것은 종래의 기술과 대조적이고, 여기서는 리맵핑 명령이 실행된 후에, 리맵핑이 다음의 모든 명령에 적용되고, 즉 소망했던 다수의 명령이 선택될 수 없다. 본 발명은 특히 바람직하게 명령 루프를 반복하도록 배치된 장치 내에 사용되고, 이 명령 루프는 상기 사전에 선택된 명령 세트를 포함한다. 이 경우에, 반복명령을 처리하기 위해 사용된 루프 하드웨어는 명령 루프가 반복될 때마다 레지스터 리맵핑 논리를 갱신하도록 배치될 수 있으므로, 레지스터 리맵핑 논리를 구성하기 위해 사용된 리맵핑 명령은 반복명령이 실행되기 전에 오직 한 번만 실행된다.

대표도

도5

색인어

데이터 처리장치, 레지스터, 명령루프, 오퍼랜드, 루프 하드웨어, 반복명령

명세서

본 발명은 데이터 처리장치에서의 레지스터 어드레싱에 관한 것으로, 특히 그것의 사용이 디지탈 신호처리에 제한되지 않더라도, 디지탈 신호처리의 분야에 관한 것이다.

전형적으로, 마이크로프로세서와 같은 데이터 처리장치는 메모리로부터 판독된 데이터 항목에 수신된 명령들을 인가하도록 배치되어 있다. 프로세서 코어는 명령을 프로세스하기 위해 마이크로프로세서 내에 설

치되고, 일반적으로 이 마이크로프로세서는 데이터 항목이 프로세서 코어에 의해 요구되기 전에 이들 데이터 항목이 다시 로드되는 복수개의 레지스터를 가질 것이다. 복수의 명령이 마이크로프로세서에 의해 수신되기 때문에, 이들 명령은 전형적으로 처리를 위한 프로세서 코어에 전달되기 전에 명령 디코더를 통과할 것이다. 다음에 이 프로세서 코어는 그것의 레지스터로부터 판독된 관련 데이터 항목에 디코드된 명령을 인가할 것이다. 그 다음에 프로세스된 명령의 결과는 이들 레지스터 중 하나로 되돌아가서 기록될 수 있거나, 또는 캐시 메모리와 같은 메모리에 직접 제공될 수 있다.

데이터 처리장치에 있어서는, 특정한 데이터 항목이 복수의 명령에 대하여 요구되는 경우가 종종 있다. 따라서, 이 데이터 항목은 그러한 모든 명령이 프로세스될 때까지 복수의 레지스터 중 하나에 보유되어야 하고, 이들 명령은 모두 그 데이터 항목이 위치되는 특정한 레지스터를 나타내어야 한다. 게다가, 명령의 범위(이하 명령 루프라고 칭함)가 복수회 반복되고, 어떤 데이터 항목들이 명령 루프 내의 다른 명령에 의해 재이용되는 것은 일반적인 것이다. 이것은 특히 디지탈 신호처리장치에서는 일반적인 것이고, 여기서 블록 필터 알고리즘과 같은 알고리즘은 1세트의 데이터 항목에 인가되기 위해 복수회 반복되어야 하는 명령 루프를 포함한다. 어떤 특정한 데이터 항목은 명령 루프 내의 다른 명령에 의해 복수회 이용될 것이다.

이 명령 루프는, 디지탈 신호처리장치가 실시간 출력 데이터 스트림을 생성하기 위해 대량의 데이터에 대하여 수행하도록 요구되는 비교적 복잡한 연산 및 논리동작에 널리 사용된다. 디지탈 신호처리기술에 대한 대표적인 애플리케이션은 아날로그 오디오 신호와 전송을 위해 부호화된 디지탈 신호 사이에서 실시간 변환을 해야 하는 이동전화를 포함한다.

명령은 대표적으로 이 명령에 의해 요구된 데이터 항목들을 포함하는 복수의 레지스터를 식별하는 1개 또는 그 이상의 오퍼랜드를 포함할 것이다. 예컨대, 대표적인 명령은 이 명령에 의해 요구된 2개의 데이터 항목을 식별하는 2개의 소스 오퍼랜드 및 1개의 수신지 오퍼랜드와, 명령의 결과가 위치되어야 하는 레지스터를 포함할 것이다. 명령이 상술한 바와 같이 1세트의 데이터 항목에 반복해서 인가되어야 할 명령 루프의 일부를 형성하는 경우에 있어서, 각 명령 오퍼랜드가 명령이 실행될 때마다 정확한 레지스터를 나타내어야 한다. 예컨대, 4개의 데이터 항목이 레지스터 내에 로드되고, 명령 루프의 일부로서, 어떤 한 개의 명령이 이들 4개의 데이터 항목의 각각에 인가되어야 한다면, 이 명령은 명령 루프 내에서 4번 재생되어야 하고, 이 명령의 오퍼랜드는 시간마다 다른 레지스터를 나타낸다. 이것은 분명히 코드 밀도에 대하여 불리한 효과를 갖는다.

마이크로프로세서 설계에 있어서, 논리 레지스터 레퍼런스의 개념을 이용하고, 이들을 실제의 물리 레지스터 레퍼런스에 맵핑하는 것에 대하여는 공지되어 있다. 예컨대, 슈퍼스칼라(superscalar) 프로세서는 종종 레지스터를 개명하며, 확실치 않고, 고장이 난 명령을 실행할 수 있다. 이 프로세서는 프로그래머의 모델보다 더 많은 레지스터를 갖고, 맵핑 테이블은 논리 레지스터를 현재의 물리적 등가물에 맵핑하기 위해 사용된다. 프로세서가 확실치 않게 복수의 명령을 실행하면, 이 프로세서는 그 결과를 보유하기 위해 많은 물리적 레지스터 수를 할당한다. 다른 명령들도 확실치 않게 이들 결과를 판독할 수 있다. 이와 같이 몇 개의 가능한 명령 '흐름'은 평행하게 수행할 것이다. 어떤 점에서, 프로세서는 일반적으로 조건 브렌치의 결과에 근거하여, 복수의 스트림 중 어느 하나가 정확한 것인가를 판정할 것이다. 이 점에서, 맵핑 테이블은 영구히 갱신되고, 어떤 레지스터의 부정확한 버전들은 폐기된다. 이 레지스터를 개명하는 목적은 명령 처리량을 증가시키는 것이다. 이것이 상기의 코드 밀도의 문제점을 해결하지 못한다는 것은 명백한 것이다.

논리 및 물리 레지스터 레퍼런스의 사용에 대한 또 다른 예로서, AMD(Advanced Micro Devices) Am29200 마이크로프로세서는 192개의 범용 레지스터를 구비한다. 이들 레지스터 중 128개는 논리 레지스터로서 지정되고, 추가 레지스터는 논리 레지스터 스택 포인터로서 지정되는데, 이 스택 포인터 레지스터는 128개의 논리 레지스터 내부로 오프셋을 제공한다. 따라서, 어떤 명령이 논리 레지스터를 나타낼 때는 언제나, 절대 레지스터 수를 계산하기 위해 스택 포인터 레지스터의 값을 이용한다. 이와 같이, 예컨대, 한 개의 명령이 논리 레지스터 1을 액세스하기를 원하고(논리 레지스터 시퀀스는 논리 레지스터 0으로 시작한다), 스택 포인터가 절대 레지스터 수 131을 나타내면(이것은 실제로 Am29200 디자인에 따른 128개의 논리 레지스터의 시퀀스에서 네번째의 레지스터이다), 이 명령은 실제로 절대 레지스터 수 132에 의해 식별된 레지스터를 액세스할 것이다.

어떤 한 개의 명령은 스택 포인터 레지스터 내의 값을 변경하기 위해 Am29200 마이크로프로세서에 대하여 실행될 수 있고, 그 후에 스택 포인터 레지스터 내의 개정된 값이 논리 레지스터를 나타내는 어떤 명령에 사용될 것이다. 이 명령에 의해 요청된 레지스터를 128개의 논리 레지스터 내의 절대 레지스터 수에 맵핑하는 것은 어떤 다른 명령이 스택 포인터 레지스터를 변경하기 위해 실행될 때까지 다음의 모든 명령에 사용될 것이다.

따라서, 상기 AMD 설계는 명령이 다른 레지스터 내의 다수의 다른 데이터 항목에 대하여 반복되어야 하는 경우에 어느 정도의 리맵핑을 허용한다는 것을 알 수 있다. 그러나, 이것은 명령의 오퍼랜드에서 식별된 레지스터가 다른 논리 레지스터에 맵핑될 때마다 개개의 명령이 스택 포인터 레지스터 내의 값을 변경하도록 공급되는 것을 요구한다. 이것에 의해, 디지탈 신호처리장치에서 대표적인 것으로서 시프트된 1세트의 데이터 항목에 대하여 명령 루프가 복수회 반복되는 경우에 중요한 오버헤드가 발생된다. 이 오버헤드는 특히 명령 루프가 소수의 명령을 포함할 때만 용인할 수 없게 된다. 이 루프가 반복되기 전에, 개개의 명령이 스택 포인터 레지스터 내의 값을 변경해야 한다. 따라서, 예컨대, 단일 명령이 데이터 항목 1-10에 인가되도록 10번 반복되면, 각 명령의 실행 전에, 개개의 명령이 스택 포인터 레지스터 내의 값을 변경해야 할 것이다. 따라서, 단일 명령을 포함하는 명령 루프는 효과적으로 2개의 명령을 갖는 명령 루프로 되고, 이것은 명확하게 명령 루프의 실행에 큰 오버헤드를 부가한다.

본 발명에 따르면, 처리되어야 할 데이터 항목을 저장하는 복수개의 레지스터와, 상기 복수개의 레지스터 내에 저장된 데이터 항목에 인가되어야 할 명령을 처리하는 프로세서와, 사전에 선택된 명령 세트 내의 논리 레지스터 레퍼런스를, 상기 프로세서에 의해 처리되어야 할 데이터 항목을 포함하는 레지스터를 식별하는 물리 레지스터 레퍼런스로 변환하는 레지스터 리맵핑 논리를 구비하는 데이터 처리장치가 제공된

다.

본 발명에 따르면, 레지스터 리맵핑 논리는 사전에 선택된 명령 세트에 대하여 레지스터 리맵핑을 행하도록 제공된다. 바람직하게는, 리맵핑 논리는 상기 사전에 선택된 명령 세트 전에 실행되는 리맵핑 명령에 의해 구성될 수 있다. 레지스터 리맵핑이 수행되어야 하는 사전에 선택된 명령 세트를 리맵핑 명령이 직접 식별할 수 있다는 것을 알 수 있다. 그러나, 바람직한 실시예에 있어서는, 리맵핑 명령에 의해 구성된 레지스터 리맵핑 논리가 명령 루프를 시작하는 등의 특정한 경우에 약간 나중에 실행될 수 있다. 그러한 경우에, 레지스터 리맵핑을 행해야 하는 사전에 선택된 명령 세트는 예컨대 사전에 선택된 세트가 명령 루프 내의 모든 명령이라고 하는 특정한 경우에 결정될 것이다.

이 방법에 의해, 리맵핑 명령은 리맵핑이 소망의 명령 세트에 적용되는 순서로 한 번만 실행되어야 한다. 이것은 AMD 마이크로프로세서의 설계와 대조되어야 하므로, 여기서 스택 포인터 레지스터 내의 값을 변경하는 명령 다음에, 동일한 리맵핑이 논리 레지스터를 나타내는 다음의 모든 명령에 인가될 것이고, 또 다른 명령은 스택 포인터 값을 변경할 때 실행되어야 하므로, 이 리맵핑은 다시 변경될 것이다.

바람직한 실시예에 있어서, 데이터 처리장치는 사전에 선택된 명령 세트로 이루어진, 반복되어야 할 명령의 범위를 정하는 반복명령과, 이 반복명령을 처리하며 레지스터 리맵핑 논리를 주기적으로 갱신하도록 배치된 루프 하드웨어를 더 구비한다. 루프 하드웨어는 전형적으로 명령의 범위가 반복될 때마다 레지스터 리맵핑 논리를 갱신하도록 배치되어, 논리 레지스터 레퍼런스에서 물리 레지스터 레퍼런스로의 서로 다른 리맵핑이 명령의 범위가 반복될 때마다 수행된다.

바람직한 실시예에 있어서, 반복명령은 레지스터 리맵핑 논리를 구성하기 위해 사용된 1개 또는 그 이상의 리맵핑 파라미터를 포함한다. 바람직하게는, 이들 리맵핑 파라미터는 반복명령에 의해 정해진 명령의 범위의 특정한 반복에 의존하여 논리 레지스터 레퍼런스에서 물리 레지스터 레퍼런스로의 리맵핑을 일으키는 것이다. 이와 같이, 레지스터 리맵핑 논리에 의해 행해진 리맵핑은 개개의 리맵핑 명령을 발행하는 것을 요구하지 않고 명령의 범위가 반복될 때마다 변경될 것이다. 작은 명령 루프에 대하여, 이것은 단일 명령으로 이루어진 명령 루프의 초기의 예를 선택하기 때문에, 극적인 성능 향상을 산출할 수 있고, 이 루프의 각 반복은 AMD 마이크로프로세서에 따른 종래의 방법을 이용하여 요구되어 왔던 2개의 명령보다는 단 한 개의 명령만을 요구한다.

반복명령이 상술한 바와 같이 1개 또는 그 이상의 리맵핑 파라미터를 포함하면, 리맵핑 명령은 상기 1개 또는 그 이상의 리맵핑 파라미터를 사용하여 레지스터 리맵핑 논리를 구성하기 위해 반복명령의 실행 전에 수행되도록 배치될 수 있지만, 상기 리맵핑 명령은 반복명령에 의해 정해진 상기 명령의 범위 내에는 포함되지는 않는다. 따라서, 리맵핑 명령은 반복명령이 실행되기 전에 한 번만 실행되어야 한다. 그 때 이 리맵핑 명령에 의해 정해진 레지스터 리맵핑 논리는 반복명령이 실행될 때 루프 하드웨어에 의해 발생될 것이다.

이 데이터 처리장치는 각각이 레지스터 리맵핑 논리에 대한 리맵핑 구성을 한정하는 적어도 한 개의 소정의 리맵핑 파라미터의 세트를 저장하는 저장수단을 구비한다. 이 경우에, 상기 반복명령 내에 포함된 상기 1개 또는 그 이상의 리맵핑 파라미터가 소정의 리맵핑 파라미터의 세트 중 하나에 대응하면, 바람직한 실시예에 있어서는, 레지스터 리맵핑 논리에 대한 대응하는 리맵핑 구성이 실행되어야 할 리맵핑 명령을 요구하지 않고 사용된다.

바람직한 실시예에 있어서는, 리맵핑 가능이 마이크로프로세서에 이용가능한 레지스터의 어떤 일정한 서브세트에 제한되는 것이 바람직하다고 하는 경우가 종종 있기 때문에, 제 1 리맵핑 파라미터는 레지스터 리맵핑 논리에 의해 리맵핑되기 쉬운 상기 복수개의 레지스터를 식별한다. 반복명령 내의 제 1 리맵핑 파라미터를 사용함으로써, 레지스터 리맵핑 논리가 소망의 레지스터의 서브세트에 제한될 수 있다.

바람직하게는, 베이스 포인터는 논리 레지스터 레퍼런스에 부가되어야 할 오프셋 값으로서 레지스터 리맵핑 논리에 의해 사용되고, 반복명령은 소정의 간격에서, 예컨대 명령의 범위가 반복될 때마다 베이스 포인터를 증가시키는 값을 식별하는 제 2 리맵핑 파라미터를 포함한다. 따라서, 반복명령 내의 제 2 리맵핑 파라미터는, 명령 루프의 각각의 반복에 대하여 논리 레지스터 레퍼런스가 레지스터 리맵핑 논리에 의해 물리 레지스터 레퍼런스에 맵핑되는 방법과 같이, 소정의 간격으로 자동 변경하기 위해 사용될 수 있다. 바람직하게는, 반복명령은 또한 제 1 증첩 값을 제공하는 제 3 리맵핑 파라미터를 포함하고, 베이스 포인터를 증가시키는 동안, 베이스 포인터가 제 1 증첩 값과 같게 되거나 초과하게 되면, 베이스 포인터의 증분은 신규 오프셋 값과 겹치게 된다. 이것에 의해 베이스 포인터의 패턴이 주기적으로 생성될 수 있다.

게다가, 이 반복명령은 제 2 증첩 값을 제공하는 제 4 리맵핑 파라미터를 포함하고, 베이스 포인터를 부가하여 형성된 레지스터 레퍼런스 및 논리 레지스터 레퍼런스가 제 2 증첩 값과 같거나 초과하면, 이 레지스터 레퍼런스는 신규 레지스터 레퍼런스와 겹치게 된다. 이 제 2 증첩 값은 실제로 리맵핑하기 쉬운 레지스터의 수를 식별하는 제 1 리맵핑 파라미터와 같게 되도록 선택되지만, 다른 값을 갖도록 선택될 수도 있다. 레지스터 리맵핑 논리가 소망의 레지스터의 개수로 억제되도록 하기 위해 이 제 2 증첩 값이 사용될 수 있다.

복수개의 레지스터는 어떤 적절한 방법으로 편성되지만, 바람직한 실시예에 있어서는, 복수개의 레지스터가 복수의 레지스터의 뱅크를 포함하고, 레지스터 리맵핑 논리는 특정한 뱅크 내의 다수의 레지스터에 대하여 리맵핑을 행하도록 배치되어 있다. 따라서, 데이터 항목은 레지스터의 한 개의 뱅크 내에 저장되지만, 예컨대 계수들은 레지스터의 또 다른 뱅크 내에 저장된다. 이 방법에 의해, 리맵핑 논리가 레지스터의 각 뱅크에 대하여 독립적으로 리맵핑을 수행하도록 배치될 수 있어, 예컨대 데이터 항목들이 계수와 관계없이 리맵핑될 수 있다.

대표적으로, 명령은 각각이 논리 레지스터 레퍼런스로 이루어진 복수의 오퍼랜드를 포함하고, 이 경우에 레지스터 리맵핑 논리는 각 오퍼랜드에 대하여 독립적으로 리맵핑을 수행하도록 배치될 수 있다.

전술한 바와 같이, 데이터 처리장치는 데이터를 처리하기 위해 사용된 어떤 장치일 수도 있지만, 바람직한 실시예에 있어서는 이 장치가 디지탈 신호처리장치이다. 디지탈 신호처리는 특히 본 발명을 이용하여

이득을 얻는 명령 루프를 이용하는데, 그 이유는 그러한 명령 루프에 대하여, 코드 밀도 및 성능이 상당히 향상될 수 있기 때문이다.

제 2 관점에서 보면, 본 발명은 데이터 처리장치를 동작시키는 방법을 제공하는데, 이 방법은 (a) 처리되어야 할 데이터 항목을 복수개의 레지스터 내에 저장하는 단계와, (b) 명령을 처리하기 위해 요구된 1개 또는 그 이상의 데이터 항목을 복수개의 레지스터로부터 검색하는 단계와, (c) 검색된 상기 1개 또는 그 이상의 데이터 항목을 이용하여 명령을 처리하는 단계를 구비하고, 상기 검색단계(b)는 사전에 선택된 명령 세트에 대하여, 상기 사전에 선택된 명령 세트 내의 논리 레지스터 레퍼런스를, 상기 처리단계(c)에서 요구된 데이터 항목을 포함하는 레지스터를 식별하는 물리 레지스터 레퍼런스로 변환하는 부가적인 단계를 구비한다.

본 발명의 실시예에 관해서는 첨부한 도면에 의거하여 설명할 것이다.

도 1은 디지탈 신호 처리장치의 구성을 나타낸 것이고,

도 2는 코프로세서의 레지스터 구성의 입력버퍼를 나타낸 것이며,

도 3은 코프로세서를 통과한 데이터경로를 나타낸 것이고,

도 4는 레지스터로부터 최고위 비트 또는 최저위 비트를 판독하는 다중화회로를 나타낸 것이며,

도 5는 바람직한 실시예에서 코프로세서에 의해 사용된 레지스터 리맵핑 논리를 나타낸 블로도이고,

도 6은 도 5에 나타낸 레지스터 리맵핑 논리를 보다 상세히 나타낸 것이며,

도 7은 블록 필터 알고리즘을 나타낸 테이블이다.

후에 설명한 시스템은 디지탈 신호처리(Digital Signal Processing: DSP)와 관련되어 있다. DSP는 다양한 형태를 취할 수 있지만, 전형적으로 큰 용량의 데이터의 고속(실시간)처리를 요구하는 처리로 간주될 것이다. 이 데이터는 전형적으로 약간의 아날로그 물리적 신호를 나타낸다. DSP의 좋은 예로서는 (전형적으로 콘볼루션동작, 변환동작, 및 상관동작을 이용하는) 아날로그의 사운드신호로의 디코딩 및 아날로그 사운드신호로부터의 인코딩을 요구하는 무선신호들이 송수신되는 디지탈 이동전화에 사용되는 것이 있다. 또 다른 예로서는 디스크 헤드로부터 재생된 신호들을 처리하여 헤드 트랙킹 제어를 하는 디스크 드라이버 제어기가 있다.

상기의 문맥으로 봐서는, 코프로세서와 협력하는 마이크로프로세서 코어(이 경우에는 Advanced RISC Machines Limited of Cambridge, United Kingdom에 의해 설계된 마이크로프로세서의 범위로부터의 ARM 코어)에 근거한 디지탈 신호 처리장치의 설명을 따르고 있다. 마이크로프로세서와 코프로세서의 인터페이스 및 그 자체의 코프로세서 구조를 명확하게 구성하여 기능적으로 DSP를 제공한다. 마이크로프로세서 코어는 ARM이라고 칭하고, 코프로세서는 피콜로(Piccolo)라고 칭한다. ARM 및 피콜로는 전형적으로 ASIC의 부분으로서 다른 소자들(예컨대, 온칩 DRAM, ROM, D/A 및 A/D 변환기 등)을 종종 포함하는 단일의 집적회로로서 제작될 것이다.

피콜로는 ARM 코프로세서이므로, ARM 명령 세트의 일부분을 실행한다. ARM 코프로세서 명령은 ARM으로 하여금 (로드 코프로세서 LDC 및 스토어 코프로세서 STC 명령을 이용하여) 피콜로와 메모리 사이에 데이터를 전송하게 하고, (코프로세서, MCR 명령으로 이동시키는 것과, 코프로세서, MRC 명령으로부터 이동하는 것을 이용하여) 피콜로로/피콜로부터 ARM 레지스터를 전송하게 한다. ARM 및 피콜로의 시너지 상호작용을 조사하는 하나의 방법은 ARM이 피콜로 데이터에 대한 강력한 어드레스 발생기로서 작용하여, 피콜로가 자유롭게 놓여 있는 상태로 큰 용량의 데이터의 실시간 처리를 요구하는 DSP 동작을 실행하여 대응하는 실시간 결과들을 생성하는 것이다.

도 1은 ARM(2)과 피콜로(4)를 나타내는데, 이 ARM(2)은 제어신호를 피콜로(4)에 제공하여 피콜로(4)로/피콜로(4)로부터의 데이터 워드의 전송을 제어한다. 명령 캐시(6)는 피콜로(4)에 의해 요구되는 피콜로 프로그램 명령 워드를 축적한다. 단일의 DRAM 메모리(8)는 ARM(2) 및 피콜로(4)에 의해 요구된 모든 데이터 및 명령 워드를 축적한다. 이 ARM(2)은 메모리(8)를 어드레싱하는 것과 모든 데이터 전송을 제어하는 것에 대하여 책임이 있다. 단 한 개의 메모리(8)와 1세트의 데이터 버스 및 어드레스 버스를 갖는 장치는 높은 버스 대역폭과 다수의 메모리 및 버스를 요구하는 전형적인 DSP 방법보다 덜 복잡하고, 비용이 많이 들지 않는다.

피콜로는 피콜로 데이터경로를 제어하는 명령 캐시(6)로부터 제 2 명령 스트림(디지탈 신호처리 프로그램 명령 워드)을 실행한다. 이들 명령은 디지탈 신호처리 형태의 동작, 예컨대 곱셈-누산과, 제어 흐름 명령, 예컨대 제로 오버헤드 루프 명령을 포함한다. 이들 명령은 피콜로 레지스터(10)에 보유되어 있는 데이터에 대하여 동작한다(도 2 참조). 이 데이터는 초기에 ARM(2)에 의해 메모리(8)로부터 전송되었다. 이 명령은 명령 캐시(6)로부터 유출되고, 명령 캐시(6)는 전 버스 마스터로서 데이터 버스를 구동한다. 작은 피콜로 명령 캐시(6)는 4라인일 것이고, 라인마다의 16워드는 맵핑된 캐시(64명령)를 나타낸다. 일부 실행에 있어서, 명령 캐시를 보다 크게 하는 것은 가치가 있다.

이와 같이 2개의 태스크는 독립하여 데이터를 로딩하는 ARM과, 그것을 처리하는 피콜로를 실행한다. 이것에 의해 16 비트 데이터에 대하여 일관된 단일 사이클의 데이터를 처리할 수 있다. 피콜로는 ARM로 하여금 연속하는 데이터를 프리페치하게 하고, 피콜로에 의해 요구되기 전에 그 데이터를 로딩하게 하는 (도 2에 나타낸) 데이터 입력 기구를 갖는다. 피콜로는 임의의 순서대로 로드된 데이터를 액세스하여, 마지막 시간에 구 데이터가 사용될 때 그것의 레지스터를 자동으로 재충전할 수 있다(모든 명령은 소스 레지스터가 재충전되어야 한다는 것을 나타내기 위해 소스 오퍼런드마다 1개의 비트를 갖는다). 이 입력 기구는 발주 버퍼라고 불려지고, 입력 버퍼(2)를 구비한다. (LDC 또는 MCR를 통해서) 피콜로 내부로 로드된 모든 값은 어느 쪽 레지스터에 대하여 값이 정해져 있는가를 나타내는 Rn 태그를 지니고 있다. 태그 Rn은 입력 버퍼 내의 데이터 워드쪽에 저장된다. 한 개의 레지스터가 레지스터 선택회로(14)를 통해서 액세스되며

명령이 데이터 레지스터가 재충전된다는 것을 나타낼 때, 이 레지스터는 신호 E를 표명함으로써 공백으로 표시된다. 레지스터는 입력버퍼(12) 내의 레지스터에 대하여 정해진 가장 오래된 로드된 데이터를 이용하여 재충전 제어회로(16)에 의해 자동으로 재충전된다. 발주 버퍼는 6개의 태그가 붙은 값을 보유하고 있다. 공간을 채우기 위해 나중에 저장된 워드가 통과된 후에 데이터 워드가 큐의 중심으로부터 추출될 수 있다는 점을 제외하고 이 입력버퍼(12)는 FIFO와 비슷한 형태를 갖는다. 따라서, 입력으로부터 가장 먼 데이터 워드가 가장 오래된 데이터 워드이고, 입력버퍼(12)가 정확한 태그 Rn를 갖는 2개의 데이터 워드를 보유하고 있을 때 레지스터를 재충전하기 위해 어느 데이터 워드가 사용되어야 하는지를 이것을 이용하여 결정할 수 있다.

피콜로는 도 3에 나타낸 바와 같이, 출력버퍼(18)(FIFO) 내에 데이터를 저장한 후에 데이터를 출력한다. 데이터는 FIFO로 순차적으로 기록되고, ARM에 의해 같은 순서로 메모리(8)에 판독된다. 출력버퍼(18)는 8개의 32비트 값을 보유하고 있다.

피콜로는 코프로세서 인터페이스를 통해서 ARM과 접속된다(도 1의 CP 제어신호). ARM 코프로세서 명령의 실행시, 피콜로는 명령을 실행하고, 그 명령을 실행하기 전에 피콜로가 준비될 까지 ARM으로 하여금 대기하게 할 수 있거나, 또는 그 명령을 실행하지 않을 수 있다. 마지막의 경우에, ARM은 정의되지 않은 명령은 제외할 것이다.

피콜로가 실행하는 대부분의 공통 코프로세서 명령은 데이터 버스를 통해서 메모리(8)에/메모리(8)로부터 각각 데이터 워드를 로드 및 저장하는 LDC 및 STC에 해당하고, ARM은 모든 어드레스를 생성한다. 이들 명령은 발주버퍼 내로 데이터를 로드하고, 출력버퍼(18)로부터의 데이터를 저장한다. 데이터 내에 로드하는데 입력 발주 버퍼에 충분한 공간이 없으면, 피콜로는 LDC으로 ARM의 기능을 정지시킬 것이고, 저장하는데 출력버퍼 내에 불충분한 데이터가 있으면, 즉 ARM이 기대하고 있는 데이터가 출력버퍼(18) 내에 없으면, STC으로 ARM의 기능을 정지시킬 것이다. 피콜로는 또한 ARM/코프로세서 레지스터 전송을 실행하여, ARM로 하여금 피콜로의 특별한 레지스터를 액세스하도록 하게 한다.

피콜로는 메모리로부터 그 자신의 명령을 페치하여 도 3에 나타낸 피콜로 데이터경로를 제어하고, 발주버퍼로부터 레지스터로 데이터를 전송하며, 또 레지스터로부터 출력버퍼(8)로 데이터를 전송한다. 이들 명령을 실행하는 피콜로의 연산 논리장치는 곱셈, 덧셈, 뺄셈, 곱셈-누산, 논리등작, 시프트 및 회전을 수행하는 곱셈/덧셈회로(20)를 갖는다. 또한, 누적/누적처분회로(22) 및 스케일/포화회로(24)가 데이터경로 내에 설치되어 있다.

피콜로 명령은 초기에 메모리로부터 명령 캐시(6) 내부로 로드되고, 이때 피콜로는 주메모리로 돌아가서 액세스할 필요없이 그 명령들을 액세스할 수 있다.

피콜로는 메모리 중단으로부터 회복할 수 없다. 따라서, 피콜로가 가상의 메모리 시스템에 사용되면, 모든 피콜로는 피콜로 태스크 동안 물리적 메모리 내에 있어야 한다. 피콜로 태스크의 실시간 특성, 예컨대 실시간 DSP가 주어지면 이것은 중요한 제한이 아니다. 메모리 중단이 발생하면, 피콜로가 정지하여 상태 레지스터 S2 내에 플래그를 설정할 것이다.

도 3은 피콜로의 전체 데이터경로 기능을 나타낸다. 레지스터 뱅크(10)는 3개의 판독포트와 2개의 기록포트를 이용한다. 1개의 기록포트(L포트)를 사용하여 발주 버퍼로부터 레지스터를 재충전한다. 출력버퍼(18)는 ALU 결과 버스(26)로부터 직접 갱신되고, 출력버퍼(18)로부터의 출력은 ARM 프로그램 제어하에 있다. ARM 코프로세서 인터페이스는 발주버퍼 내의 LDC(로드 코프로세서)명령과, 출력버퍼(18)뿐만 아니라 MCR 및 MRC(CP 레지스터로/CP 레지스터로부터 ARM 레지스터를 이동)로부터의 STC(스토어 코프로세서) 명령을 레지스터 뱅크(10)로 수행한다.

남아 있는 레지스터 포트는 ALU에 사용된다. 2개의 판독포트(A 및 B)는 곱셈/덧셈회로(20)에 대한 입력을 구동하고, C 판독포트를 사용하여 누적/누적처분회로(22) 입력을 구동한다. 남아 있는 기록포트 W를 사용하여 결과를 레지스터 뱅크(10)로 반환한다.

상기 곱셈기(20)는 임의의 48비트 누적과, 16×16 부호 또는 부호없는 곱셈을 행한다. 스케일부(24)는 임의의 포화 전에 오는 0-31의 즉시연산 또는 우측의 논리 시프트를 제공할 수 있다. 이 시프터 및 논리장치(30)는 사이클마다 시프트동작이나 논리동작 중 하나를 수행할 수 있다.

피콜로는 16개의 범용 레지스터 D0-D15 또는 A0-A3, X0-X3, Y0-Y3, Z0-Z3를 갖는다. 첫 번째의 4개의 레지스터(A0-A3)는 누산기로서 지정되며 폭이 넓은 48비트이고, 여분의 16비트는 많은 연속계산 동안 오버플로에 대하여 감시한다. 나머지 레지스터는 폭이 넓은 32비트이다.

피콜로의 레지스터의 각각은 2개의 독립된 16비트값을 포함한 것으로 간주될 수 있다. 낮은 값의 절반은 비트 0-15이고, 높은 값의 절반은 비트 16-31이다. 명령들은 소스 오퍼랜드로서 각 레지스터의 특정한 16비트 절반을 나타낼 수 있거나, 전체 32비트 레지스터를 나타낼 수도 있다.

피콜로는 또한 포화된 연산을 제공한다. 곱셈, 덧셈 및 뺄셈 명령의 변형은 포화된 결과가 수신지 레지스터의 크기보다 크면, 포화된 결과를 제공한다. 수신지 레지스터는 48비트 누산기이고, 그 값은 32비트로 포화된다(즉, 48비트값을 포화시키는 방법은 없다). 48비트 레지스터에 대하여는 어떠한 오버플로도 검출되지 않는다. 이것은 오버플로를 일으키기 위해 적어도 65536 곱셈 누산 명령을 취하기 때문에, 합당한 제한이다.

각 피콜로 레지스터는 '공백'(E 플래그, 도 2 참조)으로서 표시되거나, 또는 한 개의 값을 포함한다(레지스터의 공백 절반을 갖는 것은 불가능하다). 처음에, 모든 레지스터는 공백으로 표시된다. 각 사이클에 대하여 피콜로는 재충전 제어회로(16)를 사용하여 입력 발주버퍼로부터 한 개의 값으로 공백 레지스터 중 하나를 충전하려고 시도한다. 양자 택일로, 이 레지스터에 ALU로부터 어떤 하나의 값이 기록되면, 더 이상 '공백'으로 표시되지 않는다. ALU로부터 레지스터에 기록되는 것과 동시에 발주버퍼로부터 레지스터 내에 배치되기를 대기하고 있는 값이 있다면, 결과가 한정되지 않는다. 공백 레지스터에 대하여 판독을 행하면, 피콜로의 실행부가 정지할 것이다.

입력 발주버퍼(Input Reorder Buffer: ROB)는 코프로세서 인터페이스와 피콜로의 레지스터 뱅크 사이에 위치되어 있다. 데이터는 ARM 코프로세서 전송으로 ROB 내에 로드된다. ROB는 값이 정해져 있는 피콜로 레지스터를 나타내는 태그를 각각이 갖는 다수의 32비트값을 포함한다. 이 태그는 또한 데이터가 전체 32비트 레지스터에 전송되어야 하는지 또는 32비트 레지스터의 하부 16비트에만 전송되어야 하는지의 여부를 나타낸다. 데이터가 전체 레지스터에 대하여 정해지면, 엔트리의 하부 16비트가 타겟 레지스터의 하부 절반에 전송될 것이고, 상부 16비트는 레지스터의 상부 절반에 전송될 것이다(타겟 레지스터가 48비트 누산기일 때 확장된 부호). 데이터가 레지스터의 하부 절반에 대해서만 정해지면, 하부 16비트가 첫 번째로 전송될 것이다.

레지스터 태그는 항상 물리적 수신지 레지스터에 관하여 설명하고, 어떠한 레지스터도 리맵핑을 행하지 않는다(레지스터 리맵핑에 관해서는 아래를 참조)

사이클마다 피콜로는 다음과 같이, ROB로부터 레지스터 뱅크로 데이터 엔트리를 전송하려고 시도한다.

- ROB에서의 각 엔트리를 조사하고, 태그를 공백상태로 있는 레지스터와 비교하여, 엔트리의 일부 또는 모두로부터 레지스터로 전송을 할 수 있는지의 여부를 결정한다.

- 전송을 할 수 있는 엔트리의 세트로부터, 가장 오래된 엔트리를 선택하고, 그것의 데이터를 레지스터 뱅크에 전송한다.

- 이 엔트리의 태그를 갱신하여 엔트리가 공백이라고 표시한다. 엔트리의 일부분만을 전송했을 때, 전송된 일부분만 공백으로 표시된다.

예컨대, 이 타겟 레지스터가 완전히 공백이고, 선택된 ROB 엔트리가 전 레지스터에 대하여 정해진 데이터를 포함하면, 전체 32비트가 전송되고, 엔트리가 공백으로 표시된다. 타겟 레지스터의 하부 절반이 공백이고, ROB 엔트리가 레지스터의 하부 절반에 대하여 정해진 데이터를 포함하면, ROB 엔트리의 하부 16비트는 타겟 레지스터의 하부 절반에 전송되고, ROB의 하부 절반은 공백으로 표시된다.

어떤 엔트리에서의 데이터의 높은 16비트 및 낮은 16비트는 독립하여 전송될 수 있다. 어떠한 엔트리도 레지스터 뱅크에 전송될 수 있는 데이터를 포함하지 않으면, 그 사이클에 대하여 어떠한 전송도 하지 않는다. 아래의 표는 타겟 ROB 엔트리와 타겟 레지스터 상태의 모든 가능한 결합을 설명한다.

	타겟, Rn, 상태		
타겟 ROB 엔트리 상태	공백	낮은 절반 공백	높은 절반 공백
전 레지스터, 양쪽 절반 유효	Rn.h <- entry.h Rn.l <- entry.l 엔트리가 공백을 표시했다.	Rn.l <-entry.l entry.l가 공백을 표시했다.	Rn.h <- entry.h entry.h가 공백을 표시했다.
전 레지스터, 높은 절반 유효	Rn.h <- entry.h 엔트리가 공백을 표시했다.		Rn.h <- entry.h 엔트리가 공백을 표시했다.
전 레지스터, 낮은 절반 유효	Rn.h <- entry.l 엔트리가 공백을 표시했다.	Rn.l <- entry.l 엔트리가 공백을 표시했다.	
반 레지스터, 양쪽 절반 유효	Rn.h <- entry.l entry.l가 공백을 표시했다.	Rn.h <- entry.l entry.l가 공백을 표시했다.	
반 레지스터 높은 절반 유효	Rn.h <- entry.h 엔트리가 공백을 표시했다.	Rn.h <- entry.h 엔트리가 공백을 표시했다.	

요약하면, 2개의 반 레지스터는 ROB로부터 독립하여 재충전될 것이다. ROB에서의 데이터는 전 레지스터에 대하여 정해진 대로 표시되거나, 또는 레지스터의 하부 절반에 대하여 정해진 2개의 16비트값으로서 표시된다.

ARM 코프로세서 명령을 이용하여 데이터를 ROB 내에 로드한다. 데이터가 얼마나 ROB 내에 표시되는가는 ARM 코프로세서 명령을 이용하여 전송을 하는 것에 의존한다. ROB를 데이터로 충전하기 위해 다음의 ARM 명령을 이용할 수 있다.

```
LDP{<cond>}<16/32>          <dest>, [Rn]{!}, #<size>
LDP{<cond>}<16/32>W         <dest>, <wrap>, [Rn]{!}, #<size>
LDP{<cond>}16U              <bank>, [Rn]{!}
MPR{<cond>}                 <dest>, Rn
```

MRP{<cond>} <dest>, Rn

ROB를 구성하기 위해 다음의 ARM 명령이 제공된다.

LDPA<bank list>

COP 명령으로서 MCR, LDP가 어셈블리되는 것처럼 처음 3개는 LDC, MPR 및 MRP로서 어셈블리된다.

상기에서, <dest>는 피콜로 레지스터(A0-Z3)를 나타내고, Rn은 ARM 레지스터를 나타내며, <size>는 0이 아닌 4의 배수이어야 하는 바이트의 정수를 나타내고, <wrap>는 정수(1, 2, 3, 8)를 나타낸다. {}로 둘러싸인 필드는 선택적이다. 발주버퍼 내에 맞출 수 있는 전송을 위해, <size>는 기껏해야 32이어야 한다. 많은 상황에서, <size>는 데드록(deadlock)을 피하기 위해 이 제한보다 작을 것이다. <16/32> 필드는 로드되어 있는 데이터가 16비트 데이터로서 취급되어야 하는지 또는 32비트 데이터로서 취급되어야 하는지의 여부를 나타낸다.

주의1: 다음의 텍스트에서, LDP 또는 LDPW을 나타내면, 이것은 명령의 16비트 및 32비트 변형을 나타낸다.

주의2: 하나의 '워드'는 2개의 16비트 데이터 항목 또는 1개의 32비트 데이터 항목으로 구성되는 메모리로부터의 32비트군이다.

LDP 명령은 전 레지스터에 대하여 정해진 대로 데이터 항목을 표시하면서, 다수의 데이터 항목을 전송한다. 이 명령은 ROB 내에 워드를 삽입하면서, 메모리 내의 어드레스 Rn으로부터 <size>/4 워드를 로드할 것이다. 전송될 수 있는 워드의 개수는 다음으로 제한된다.

- 그 수량<size>은 0이 아닌 4의 배수이어야 한다.

- <size>는 특정한 실행을 위해 ROB의 크기보다 작거나 같아야 한다(제 1 버전에서는 8개의 워드, 장래의 버전에서는 제 1 버전과 마찬가지이다).

전송된 제 1 데이터 항목은 <dest>에 대하여 정해진 대로 태그가 붙어있고, 제 2 데이터 항목은 <dest>+1 등에 대하여 정해진 대로 태그가 붙어있다(Z3에서 A0까지 겹쳐 채로). !가 지정되면, 레지스터 Rn은 나중에 <size>만큼 증가된다.

LDP16 변형이 이용되면, 엔디안(endian) 특수동작은 메모리 시스템으로부터 반환되는 대로 32비트 데이터 항목을 형성하는 2개의 16비트 하프워드에 대하여 수행된다. 큰 엔디안 및 작은 엔디안 지지에 대한 보다 상세한 것에 대하여는 다음을 참조한다.

LDPW 명령은 다수의 데이터 항목을 1세트의 레지스터에 전송한다. 전송된 제 1 데이터 항목은 <dest>에 대하여 정해진 대로 태그가 붙어 있고, 다음의 데이터 항목은 dest>+1 등에 대하여 정해진 대로 태그가 붙어 있다. <wrap> 전송이 발생되면, 전송된 다음 항목은 <dest> 등에 대하여 정해진 대로 태그가 붙어있다. <wrap> 수양은 하프워드의 양으로 지정된다.

LDPW에 대하여는, 다음의 제한이 적용된다.

- 수량 <size>는 0이 아닌 4의 배수이어야 한다.

- <size>는 특정한 실행을 위해 ROB의 크기보다 작거나 같아야 한다(제 1 버전에서는 8개의 워드, 장래의 버전에서는 제 1 버전과 마찬가지이다).

-<dest>는 {A0, X0, Y0, Z0} 중 하나일 것이다.

-<wrap>는 LDP32W에 대하여 {2, 4, 8} 하프워드 중 한 개일 것이고, LDP16W에 대하여는 {1, 2, 4, 8} 중 하나일 것이다.

- 수량 <size>는 2*<wrap>보다 크지 않으면, 어떠한 중첩(wrap)도 발생하지 않고, 대신 LDP명령이 사용될 것이다.

예컨대, 아래의 명령은,

```
LDP32W      X0, 2, [R0]!, #8
```

ROB 내로 2개의 워드를 로드하여, 전 레지스터 X0에 대하여 정해진 대로 2개의 워드를 표시할 것이다. R0은 8만큼 증가될 것이다. 아래의 명령은,

```
LDP32W      X0, 4, [R0], #16
```

ROB 내로 4개의 워드를 로드하여, X0, X1, X0, X1에 대하여 정해진 대로(이 순서대로) 4개의 워드를 표시할 것이다. R0은 영향을 받지 않을 것이다.

LDP16W에 대하여, <wrap>는 1, 2, 4 또는 8로서 지정될 것이다. 1의 중첩에 의해 모든 데이터는 수신지 레지스터 <dest>.1의 하부 절반에 대하여 정해진 대로 태그가 붙어있다. 이것은 '반 레지스터' 경우이

다.

예컨대, 아래의 명령은,

```
LDP16W          X0,1,[R0]!,#8
```

ROB 내로 2개의 워드를 로드하여, X0,1에 대하여 정해진 16비트 데이터로서 2개의 워드를 표시할 것이다. R0은 8만큼 증가될 것이다. 아래의 명령은,

```
LDP16W          X0,4,[R0],#6
```

데이터가 메모리로부터 반환될 때 데이터에 대하여 엔디안 특수동작을 수행한다는 사실을 제외하고, LDP32W 예와 비슷한 방식으로 동작할 것이다.

LDP 명령의 사용하지 않은 모든 인코딩은 앞으로의 확장을 위해 지정될 것이다.

LDP16U 명령을 제공하여 워드가 아닌 정렬된 16비트 데이터의 효율적인 전송을 지지한다. LDP16U 지지는 레지스터 D4-D15에 대하여 제공된다(X, Y 및 Z 뱅크). LDP16U 명령은 메모리로부터 피콜로 내로 데이터의 1개의 32비트 워드(2개의 16비트 데이터 항목)를 전송할 것이다. 피콜로는 이 데이터의 하부 16비트를 폐기하고, 유지 레지스터 내에 상부 16비트를 저장할 것이다. X, Y 및 Z 뱅크에 대해서는 유지 레지스터가 있다. 뱅크의 유지 레지스터가 일단 준비되면, 데이터가 이 뱅크 내의 레지스터에 대하여 정해질 때 LDP{W} 명령의 작용을 변경한다. ROB 내에 로드된 데이터는 LDP 명령에 의해 전송되어 있는 데이터의 하부 16비트와 유지 레지스터의 연결에 의해 형성된다. 전송되어 있는 데이터의 상부 16비트는 유지 레지스터 내에 놓여 있다.

```
entry <- data.l : holding_resister
holding_register <- data.h
```

이 동작의 모드는 LDPA 명령에 의해 오프될 때까지 불변이다. 이 유지 레지스터는 수신지 레지스터 태그 또는 사이즈를 기록하지 않는다. 이들 특성은 data.l의 다음 값을 제공하는 명령으로부터 획득된다.

엔디안의 특수동작은 항상 메모리 시스템에 의해 반환된 데이터에 대하여 발생한다. 모든 32비트 데이터 항목을 메모리 내에 정렬된 워드라고 가정하기 때문에, 16비트가 LDP16U와 동등하지 않은 것은 없다.

LDPA 명령을 사용하여 LDP16U 명령에 의해 시작된 동작의 정렬되지 않은 모드를 전환한다. 이 정렬되지 않은 모드는 뱅크 X, Y, Z에 대하여 독립하여 턴오프될 것이다. 예컨대, 다음의 명령은,

```
LDPA            {X, Y}
```

뱅크 X, Y에 대하여 정렬되지 않은 모드를 턴오프할 것이다. 이들 뱅크의 유지 레지스터 내의 데이터는 폐기될 것이다.

정렬되지 않은 모드에 없는 뱅크에 대하여 LDPA를 실행하는 것을 허용하여, 정렬되지 않은 모드에 그 뱅크를 남겨 놓을 것이다.

MPR 명령은 피콜로 레지스터<dest>에 대하여 정해진 대로, ROB 내에 ARM 레지스터 Rn의 내용을 배치한다. 수신지 레지스터<dest>는 A0-Z3의 범위 내의 어떤 전 레지스터일 것이다. 예컨대, 다음의 명령은,

```
MPR             X0, R3
```

ROB 내에 R3의 내용을 전송하여, 전 레지스터 X00에 대하여 정해진 대로 데이터를 표시할 것이다.

ARM은 내부에 거의 엔디안이 없기 때문에 ARM으로부터 피콜로로 데이터가 전송될 때 어떠한 엔디안 특수동작도 발생하지 않는다.

MPRW 명령이 16비트 피콜로 레지스터<dest..1에 대하여 정해진 2개의 16비트 데이터 항목으로서 데이터를 표시하면, ARM 레지스터 Rn의 내용을 ROB 내에 배치한다. <dest>에 대한 제한은 LDPW 명령(즉, A0, X0, Y0, Z0)에 대한 것과 동일하다. 예컨대, 다음의 명령은,

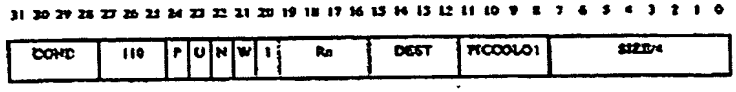
```
MPRW            X0, R3
```

ROB 내로 R3의 내용을 전송하여, X0.1에 대하여 정해진 2개의 16비트의 양으로서 데이터를 표시할 것이

다. 1의 증첩과 LDP16W 경우에 관하여는, 32비트 레지스터의 하부 절반만 목표로 정해질 것이다.

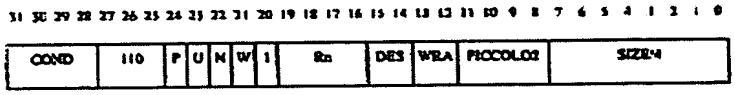
MPR에 관해서는, 어떠한 엔디안 특수동작도 데이터에 적용되지 않는다.

LDP는 다음과 같이 인코드된다.



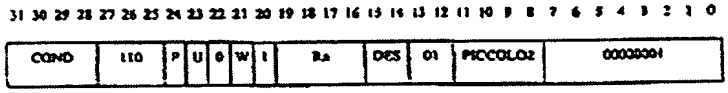
여기서, PICCOLO1은 피콜로의 첫 번째 코프로세서 수(현재 8)이다. N비트는 LDP32(1)와 LDP16(0) 중 어느 하나를 선택한다.

LDPW는 다음과 같이 인코드된다.



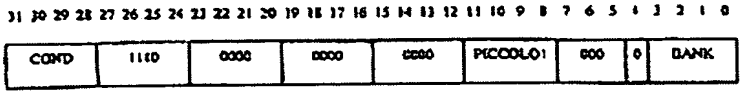
여기서 DEST는 수신지 레지스터 A0, X0, Y0, Z00에 대한 0-30이고, WRAP는 증첩값 1, 2, 4, 8에 대한 0-30이다. PICCOLO2는 피콜로의 제 2 코프로세서 수(현재 9)이다. N비트는 LDP32(1)와 LDP16(0) 중 어느 하나를 선택한다.

LDP16U는 다음과 같이 인코드된다.



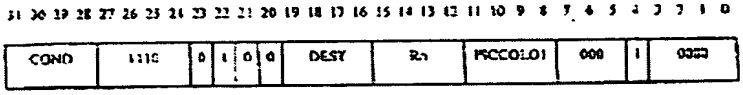
여기서, DEST는 수신지 뱅크 X, Y, Z의 1-30이다.

LDPA는 다음과 같이 인코드된다.

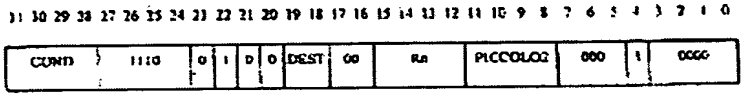


여기서, BANK[3:0]를 사용하여 뱅크 기준으로 정렬되지 않은 모드를 턴오프한다. BANK[1]이 설정되면, 뱅크 X에 대한 정렬되지 않은 모드가 턴오프된다. BANK[2]와 BANK[3]이 각각 설정되면, 뱅크 Y, Z에 대한 정렬되지 않은 모드를 턴오프한다. 이것은 COP 동작이다.

MPR은 다음과 같이 인코드된다.



MPRW는 다음과 같이 인코드된다.



여기서, DEST는 수신지 레지스터 X0, Y0, Z00에 대한 1-30이다.

출력 FIFO는 8개의 32비트값까지 유지할 수 있다. 이들은 다음의 (ARM) 연산코드 중 하나를 사용하여 피콜로로부터 전송된다.

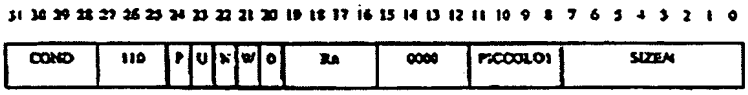
STP{<cond>}<16/32> [Rn]{!}, #<size>

MRP Rn

!이 존재하면, 출력 FIFO로부터 Rn을 색인 지정하는 ARM 레지스터 Rn에 의해 주어진 어드레스로 <size>/4 워드를 첫 번째로 세이브한다. 데드록(deadlock)을 방지하기 위해, <size>는 출력 FIFO의 사이즈보다 커야 한다(이 실행시에는 8개의 엔트리). STP16 변형이 사용되면, 엔디안 특수동작이 메모리 시스템으로부터 반환된 데이터에 발생할 것이다.

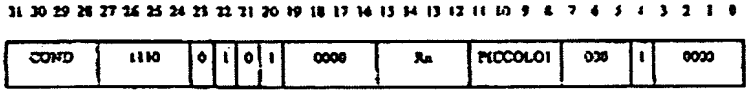
MRP 명령은 출력 FIFO로부터 1개의 워드를 제거하고, 그것을 ARM 레지스터 Rn에 배치한다. MPR에 관해서는, 어떠한 엔디안 특수동작도 데이터에 적용되지 않는다.

STP에 대한 ARM 인코딩은 다음과 같다.



여기서, N은 STP32(1)와 STP16(0) 중 하나를 선택한다. P, U 및 W비트의 정의를 위해, ARM 데이터 시트에 관하여 설명한다.

MRP에 대한 ARM 인코딩은 다음과 같다.



피콜로 명령 세트는 내부에 엔디안 동작이 거의 없다고 가정한다. 예컨대, 16비트 절반으로서 32비트 레지스터를 액세스할 때, 하부 절반은 비트 15-0을 점유한다고 가정한다. 피콜로는 큰 엔디안 메모리를 가진 시스템 및 주변에서 동작하므로, 정확한 방법으로 16비트 짝 찬 데이터를 로드하도록 처리해야 한다.

ARM(예컨대, Advanced RISC Machines Limited of Cambridge, United Kingdom에 의해 제조된 ARM7 마이크로프로세서)과 같은 피콜로(즉, DSP 제조 코프로세서)는 프로그래머가 아마도 프로그래머블 주변장치로 제어할 수 있는 'BIGEND'구성 핀을 갖는다. 피콜로는 이 핀을 이용하여 입력 발주버퍼 및 출력 FIFO를 구성한다.

ARM가 짝 찬 16비트 데이터를 발주버퍼 내로 로드할 때, LDP 명령의 16비트 형태를 사용하여 16비트 데이터를 표시한다. 이 정보는 'BIGEND' 구성 입력의 상태와 결합되어 데이터를 유지 레치 및 발주버퍼 내에 적당한 순서로 배치한다. 특히, 큰 엔디안 모드에서, 유지 레지스터는 로드된 워드의 하부 16비트를 저장하고, 다음 로드의 상부 16비트와 짝이 된다. 유지 레지스터 내용은 항상 발주버퍼 내로 전송된 워드의 하부 16비트에서 끝난다.

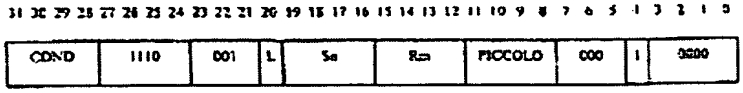
출력 FIFO는 짝 찬 16비트 또는 32비트 중 어느 하나를 포함한다. 프로그래머는 STP 명령의 정확한 형태를 이용해야 하기 때문에, 피콜로는 16비트 데이터가 데이터 버스의 정확한 절반에 대하여 제공된다는 것을 확보할 수 있다. 큰 엔디안으로 구성되면, 상부 및 하부 16비트 절반은 STP의 16비트 형태가 사용될 때 교환된다.

피콜로는 오직 ARM으로부터 액세스될 수 있는 4개의 전용 레지스터를 갖는다. 이들 전용 레지스터는 S0-S2라고 칭한다. 이들 전용 레지스터는 MRC 및 MCR 명령으로만 액세스될 수 있다. 연산코드는 다음과 같다.

MPSR Sn, Rm

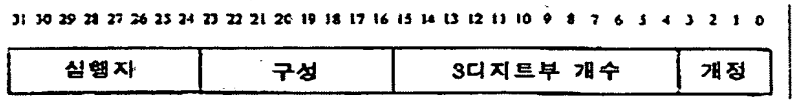
MRPS Rm, Sn

이들 연산코드는 ARM 레지스터 Rm과 전용 레지스터 Sn 사이에 32비트 값을 전송한다. 이들 연산코드는 코프로세서 레지스터 전송으로 ARM에서 다음과 같이 인코드된다.



여기서, L은 MPSR에 대하여는 00이고, MRPS에 대하여는 10이다.

레지스터 S0은 피콜로 유일한 ID 및 개정 코드를 포함한다.



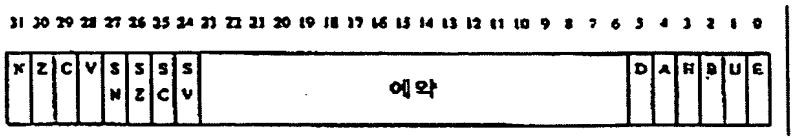
비트[3:0]은 프로세서에 대한 개정 수를 포함한다.

비트[15:4]는 2진 코드화된 10진 형식: 피콜로용 0×500에서 3디지트부의 수를 포함한다.

비트[23:16]은 구성버전: 0×00=버전 1을 포함한다.

비트[31:24]는 실행자 트레이드표시: 0×41=A=ARM Ltd의 ASCII 코드를 포함한다.

레지스터 S1은 피콜로 상태 레지스터이다.



1차 조건 코드 플래그(N, Z, C, V)

2차 조건 코드 플래그(SN, SZ, SC, SV)

E 비트: 피콜로는 ARM에 의해 기능이 억제되어, 정지되었다.

U 비트: 피콜로는 UNDEFINED 명령과 흡사하여 정지되었다.

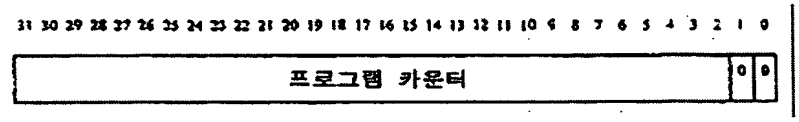
B 비트: 피콜로는 BREAKPOINT과 흡사하여 정지되었다.

H 비트: 피콜로는 HALT 명령과 흡사하여 정지되었다.

A 비트: 피콜로는 메모리 중단(로드, 저장 또는 피콜로 명령)을 허용하여 정지되었다.

D 비트: 피콜로는 데드록 조건을 검출하여 정지되었다(이하 참조).

레지스터 S2는 다음과 같은 피콜로 프로그램 카운터이다.



프로그램 카운터에 기록하는 경우에 그 어드레스에서 프로그램을 실행하는 피콜로를 게시할 것이다. 피콜로는 프로그램 카운터에 기록함으로써 항상 게시되기 때문에, 리세트시 프로그램 카운터는 규정되지 않는다.

실행시, 피콜로는 명령의 실행과 코프로세서 인터페이스의 상태를 감시한다. 그것이 검출되면,

- 피콜로는 이용가능한 엔트리를 갖기 위해 재충전되어야 할 레지스터 또는 출력 FIFO 중 어느 하나를 기다리지 않는다.

- 코프로세서 인터페이스는 ROB에서의 불충분한 공간 또는 출력 FIFO에서의 불충분한 항목 때문에 사용중 대기하고 있다.

이들 조건의 양쪽이 검출되면, 피콜로는 그것의 상태 레지스터에서 D비트를 설정하고, ARM 코프로세서 명령을 정지 및 거부하여, ARM으로 하여금 불확정의 명령을 트랩하게 한다.

ARM 및 피콜로 프로그램 카운터 및 레지스터를 판독함으로써, 데드록 조건의 검출에 의해 조건이 발생되었다는 것을 프로그래머에게 적어도 경고할 수 있고, 정확한 고장점을 보고할 수 있는 시스템이 구성될 수 있다. 데드록은 오직 부정확한 프로그램 또는 피콜로의 상태를 변조하는 시스템의 또 다른 부분으로 인해 발생할 수 있다. 데드록은 데이터 부족 또는 '오버로드'로 인해서는 발생할 수 없다.

ARM으로부터 피콜로를 제어하기 위해 사용되는 이용가능한 몇몇 동작이 있고, 이들 동작은 COP 명령에 의해 제공된다. 이들 COP 명령은 오직 ARM이 특권상태에 있을 때만 인정될 것이다. 이 경우가 아니면, 피콜로는 불확정의 명령 트랩을 취하는 ARM에서 생긴 COP 명령을 거부할 것이다.

다음의 동작들이 이용가능하다.

_ 리세트

_ 상태 액세스 모드 입력

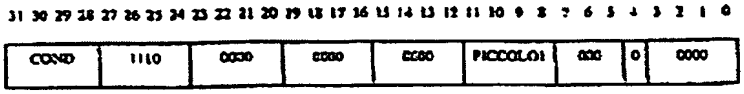
_ 인에이블

_ 디스에이블

피콜로는 PRESET 명령을 사용하여 소프트웨어에서 리세트될 것이다.

PRESET ; 피콜로 상태를 클리어

이 명령은 다음과 같이 인코드된다.



이 명령이 실행되면, 아래의 경우가 발생한다.

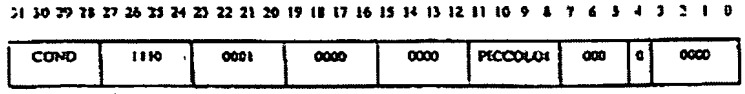
- 모든 레지스터는 공백으로 표시된다(재충전을 준비).
- 입력 ROB가 클리어된다.
- 출력 FIFO은 클리어된다.
- 부프 카운터는 리세트된다.
- 피콜로는 정지된 상태로 있다(그리고 S2의 H 비트가 설정될 것이다).

PRESET 명령을 실행함으로써 몇몇 사이클이 완성될 것이다(이 실시예에 대하여는 2-3). 명령이 실행되고 있는 동안, 피콜로에 대하여 실행되어야 할 다음의 ARM 코프로세서 명령이 사용중 대기될 것이다.

상태 액세스 모드에서, 피콜로의 상태는 STC 및 LDC 명령을 사용하여 세이브 및 재저장될 것이다(ARM으로부터 피콜로 상태를 액세스하는 것에 관하여는 아래를 참조). 상태 액세스 모드를 입력하기 위해, PSTATE 명령이 첫 번째로 실행되어야 한다.

PSTATE; 상태 액세스 모드 입력

이 명령은 다음과 같이 인코드된다.



명령이 실행되면, PSTATE 명령은,

- 피콜로(아직 정지되지 않으면)를 정지하여, 피콜로의 상태 레지스터에서 E비트를 설정할 것이다.
- 피콜로를 그것의 상태 액세스 모드로 구성할 것이다.

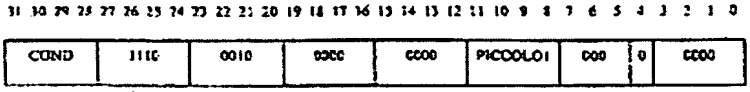
피콜로가 정지할 수 있기 전에 피콜로의 명령 파이프라인이 제거되어야 하기 때문에 PSTATE 명령을 실행함으로써 몇몇 사이클이 완성될 것이다. 명령이 실행되면, 피콜로에 대하여 실행되어야 할 다음의 ARM 코프로세서 명령이 사용중 대기될 것이다.

PENABLE 및 PDISABLE 명령은 빠른 콘텍트 전환을 위해 사용된다. 피콜로가 디스에이블되면, 전용 레지스터 0, 1(ID 및 상태 레지스터)만 특권모드로부터 액세스될 수 있다. 어떤 다른 상태로의 액세스 또는 사용자 모드로부터의 어떤 액세스는 ARM 불확정의 명령을 제외시킨다. 피콜로를 디스에이블할 때 피콜로가 실행을 정지한다. 피콜로가 실행을 정지했을 때, 피콜로는 상태 레지스터에서 E비트를 설정함으로써 그 사실을 인정할 것이다.

피콜로는 PENABLE 명령을 실행함으로써 인에이블된다.

PENABLE; 피콜로 인에이블

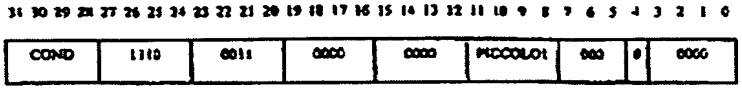
이 명령은 다음과 같이 인코드된다.



피콜로는 PDISABLE 명령을 실행함으로써 디스에이블된다.

PDISABLE ; 피콜로 디스에이블

이 명령은 다음과 같이 인코드된다.



이 명령이 실행되면, 아래의 경우가 발생한다.

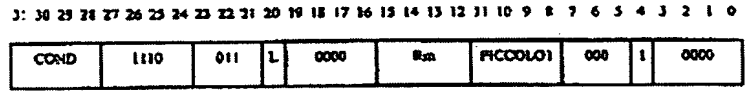
- 피콜로의 명령 파이프라인이 제거될 것이다.
- 피콜로가 정지할 것이고, 상태 레지스터 내의 H비트가 설정될 것이다.

피콜로 명령 캐시는 피콜로 데이터경로를 제어하는 피콜로 명령을 보유한다. 이것이 존재하면, 이것은 적어도 64개의 명령을 보유하여, 16개의 워드 경계선에서 시작할 것을 보증한다. 그것의 작용에 의해 캐시가 지정된 어드레스에서 시작하는 (16)명령의 라인을 페치하였다. 이 페치는 캐시가 이미 이 어드레스와 관련된 데이터를 보유하더라도 발생한다.

PMIR Rm

피콜로는 PMIR이 수행될 수 있기 전에 정지되어야 한다.

이 연산코드의 MCR 인코딩은 다음과 같다.

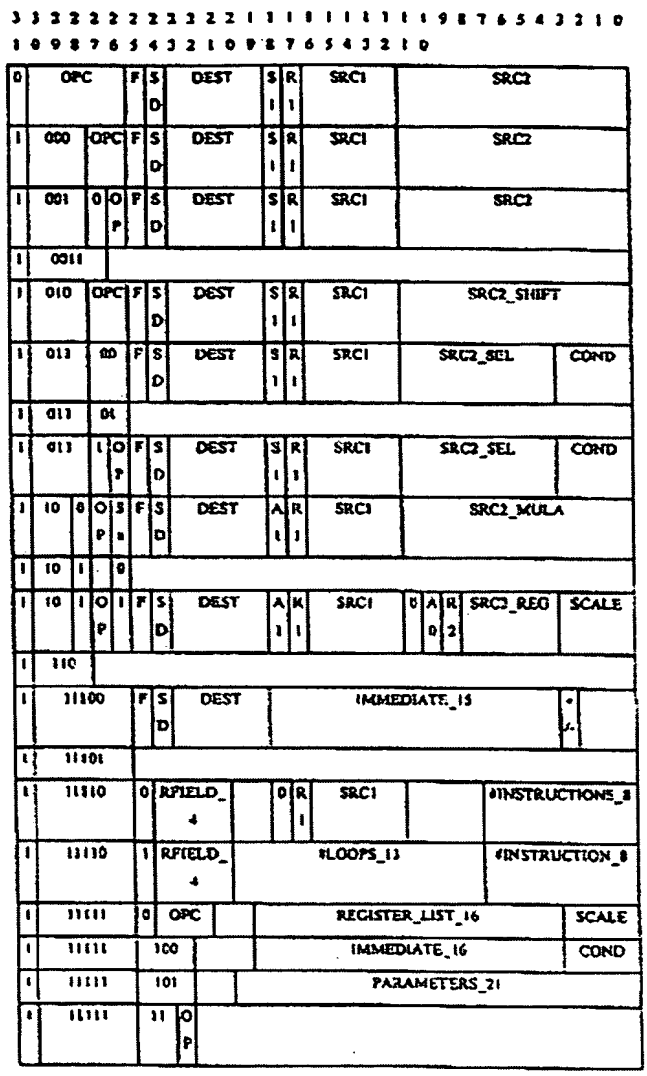


이 부분은 피콜로 데이터 경로를 제어하는 피콜로 명령 세트에 관하여 논의한다. 각 명령은 32비트이다. 이 명령은 피콜로 명령 캐시로부터 판독된다.

명령 세트의 디코딩은 완전히 일직선으로 진행한다. 상부 6비트(26-31)는 약간의 특수명령에 대하여 중요치 않은 연산코드를 제공하는 비트 22-25를 중요한 연산코드에 준다. 회색으로 음영을 나타낸 비트는 현재 확장을 위해 사용되지 않고 지정된다(이들 비트는 지시된 값을 포함해야 한다).

7개의 중요한 명령 클래스가 있다. 이것은 약간의 부분집합을 디코딩하는 것을 용이하게 하기 위해, 명령

에서의 주 연산코드 필드에 완전히 대응하지 않는다.



상기 표에서의 명령은 다음의 명칭을 갖는다.

표준 데이터 연산

논리연산

조건 덧셈/뺄셈

불확정

시프트

선택

불확정

병렬 선택

곱셈 누적

불확정

이중 곱셈

불확정

부호 즉시 이동

불확정

반복

반복

레지스터 리스트 연산

브렌치

개명 파라미터 이동

정지/중단

명령의 각 클래스에 대한 형식은 다음 부분에서 상세히 설명된다. 소스 및 수신지 오퍼랜드 필드는 레지스터 리맵핑 그대로, 대부분 명령에 공통이고, 나눗셈 부분에서 상세히 설명된다.

대부분 명령은 2개의 소스 오퍼랜드, 즉 소스 1 및 소스 2를 요구한다. 일부 예외는 절대적 과잉 공급하고 있다.

소스 1(SRC1) 오퍼랜드는 다음의 7비트 형식을 갖는다.

18	17	16	15	14	13	12
사이즈	재충전	레지스터 수				Hi/Lo

필드의 소자들은 다음의 의미를 갖는다.

-사이즈-는 판독을 위한 오퍼랜드의 크기를 나타낸다(1=32비트, 0=16비트).

-재충전-은 레지스터가 판독된 후에 공백으로 표시되어야 하고, ROB로부터 재충전될 수 있다는 것을 지정한다.

-레지스터 수-는 판독을 위한 16개의 32비트 레지스터 중 어느 것인가를 인코드한다.

-Hi/Lo-는 16비트 판독을 위해 판독을 위한 32비트 레지스터의 절반을 나타낸다. 32비트 오퍼랜드에 대하여, 설정되면, 2개의 16비트 레지스터의 절반이 서로 교환되어야 한다는 것을 나타낸다.

사이즈	Hi/Lo	액세스된 레지스터의 부분
0	0	낮은 16비트
0	1	높은 16비트
1	0	전 32비트
1	1	전 32비트, 절반을 교환했다.

레지스터 사이즈는 레지스터 번호: 낮은 16비트에 대해서는 .l, 높은 16비트에 대해서는 .h, 상호 교환된 상부 및 하부 16개의 비트를 가진 32비트에 대해서는 .x에 접미사를 부가함으로써 어셈블러로 지정된다.

일반적인 소스 2(SRC2)는 다음의 3개의 12비트 형식 중 하나를 갖는다.

11	10	9	8	7	6	5	4	3	2	1	0
0	S2	R2	레지스터 수				Hi/Lo	SCALE			
1	0	ROT	IMMED_8								
1	1	IMMED_6						SCALE			

도 4는 선택된 레지스터의 적당한 절반을 피연로 데이터경로로 전환하기 위해 Hi/Lo 비트 및 사이즈 비트에 응답하는 멀티플렉서 배치를 나타낸다. 이 사이즈 비트가 16비트를 나타내면, 부호 확장회로는 부호는 0 또는 1로 데이터경로의 최고위 비트를 매립한다.

첫 번째의 인코딩은 소스를 레지스터라고 지정하고, 그 필드는 SRC1 규칙자(specifier)와 같은 인코딩을 갖는다. SCALE 필드는 ALU의 결과에 적용되어야 할 스케일을 지정한다.

SCALE				액션
3	2	1	0	
0	0	0	0	ASR #0
0	0	0	1	ASR #1
0	0	1	0	ASR #2
0	0	1	1	ASR #3
0	1	0	0	ASR #4
0	1	0	1	RESERVED
0	1	1	0	ASR #6
0	1	1	1	ASL #1
1	0	0	0	ASR #8
1	0	0	1	ASR #16
1	0	1	0	ASR #10
1	0	1	1	RESERVED
1	1	0	0	ASR #12
1	1	0	1	ASR #13
1	1	1	0	ASR #14
1	1	1	1	ASR #15

순환 인코딩의 8비트 즉치는 8비트 값과 2비트 순환으로 표시할 수 있는 32비트 즉치를 발생할 수 있다. 다음의 표는 8비트 값 XY로부터 발생될 수 있는 즉치를 나타낸다.

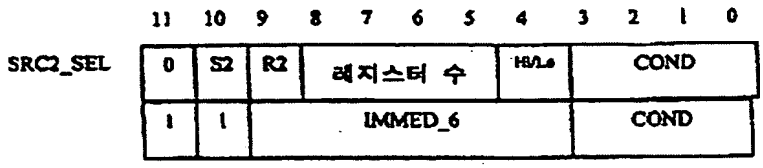
순환	즉치
0	0x000000XY
1	0x0000XY00
10	0x00XY0000
11	0xXY000000

6비트 즉치 인코딩은 ALU의 출력에 적용된 스케일과 함께, 6비트의 부호없는 즉치(범위 0-63)를 사용할 수 있다.

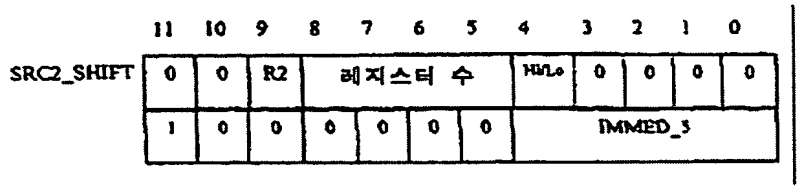
일반적인 소스 2 인코딩은 대부분 명령 변형에 공통이다. 제한된 소스 2 인코딩의 서브세트를 지지하거나, 그것을 조금 변형하는 규칙에는 몇몇 예외가 있다.

- 선택 명령.
- 시프트 명령.
- 병렬 연산.
- 곱셈 누적 명령.
- 곱셈 이중 명령.

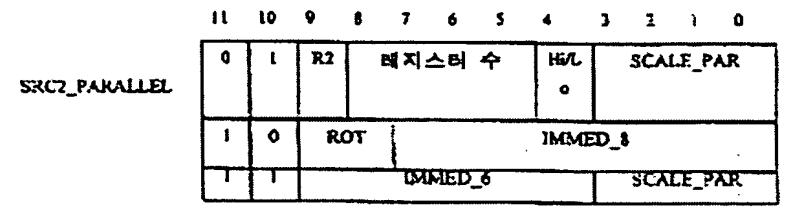
선택 명령은 오직 한 개의 레지스터 또는 8비트의 부호없는 즉치인 오퍼런드를 지지한다. 이들 비트가 명령의 조건 필드에 의해 사용되기 때문에 이 스케일은 이용 불가능하다.



시프트 명령은 오직 16비트 레지스터 또는 1과 31사이의 5비트의 부호없는 즉치인 오퍼런드를 지지한다. 그 결과의 어떠한 스케일도 이용할 수 없다.



병렬 연산의 경우에 있어서, 레지스터가 오퍼런드의 소스로서 지정되면, 32비트 판독이 수행되어야 한다. 이 즉시 인코딩은 병렬 연산에 대하여 약간 다른 의미를 갖는다. 이것에 의해 즉치가 32비트 오퍼런드의 양쪽 16비트 절반 위에 복사 될 수 있다. 약간 제한된 스케일의 범위는 병렬 연산에 대하여 이용가능하다.

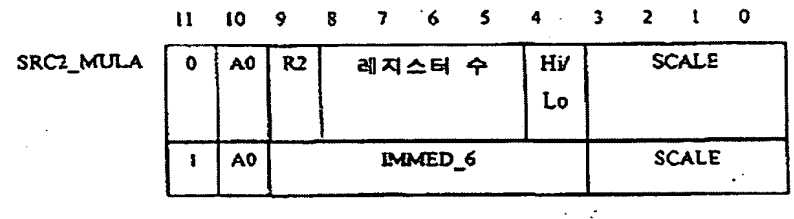


6비트 즉치가 사용되면, 이것은 항상 32비트 양의 양쪽 절반 위에 복사된다. 8비트 즉치가 사용되면, 8비트 즉치가 32비트 양의 상부 절반 위에서 순환되어야 한다는 것을 이 순환이 나타내는 경우에만 8비트 즉치가 복사된다.

순환	즉치
0	0x000000XY
1	0x0000XY00
10	0x00XY00XY
11	0xXY00SY00

병렬 선택 연산에 대하여는 어떠한 스케일도 이용할 수 없고, 이 스케일 필드는 이들 명령에 대하여 0으로 설정될 것이다.

곱셈 누적 명령에 의해 8비트 순환 즉치가 지정될 수 없다. 필드의 비트 10을 사용하여 어떤 누산기를 부분적으로 사용할 것인가를 지정한다. 소스 2는 16비트 오퍼런드로서 암시된다.



곱셈 이중 명령은 정수를 사용할 수 없다. 16비트 레지스터만 지정될 수 있다. 필드의 비트 10을 사용하

여 어느 누산기를 사용할 것인가를 부분적으로 지정한다.

	11	10	9	8	7	6	5	4	3	2	1	0
SRC2_MULD	0	A0	R2	레지스터 수				Hi/Lo	SCALE			

일부 명령은 항상 32비트 연산(예컨대, ADDADD)을 포함하고, 이를 경우에 사이즈 비트는 1로 설정될 것이며, Hi/Lo 비트는 32비트 오퍼랜드의 2개의 16비트 절반을 선택적으로 교환하기 위해 사용된다. 일부 명령은 항상 16비트 연산(예컨대, MUL)을 포함하고, 사이즈 비트는 0으로 설정되어야 한다. Hi/Lo는 어떤 반 레지스터가 사용되는가를 선택한다(결여된 사이즈 비트가 1로 디폴트된다고 가정한다). 곱셈 누산 명령은 소스 누산기 및 수신지 레지스터의 독립적인 지정을 허용한다. 이들 명령에 대하여, 사이즈 비트를 사용하여 소스 누산기를 표시하고, 사이즈 비트는 명령 형태에 의해 0으로서 암시된다.

16비트값이 (A 또는 B 버스를 통해서) 판독되면, 이것은 자동으로 32비트 양으로 부호 확장된다. 48비트 레지스터가 (A 또는 B 버스를 통해서) 판독되면, 하부 32비트만이 버스 위에 나타난다. 따라서, 모든 경우에 소스 1 및 소스 2는 32비트 값으로 변환된다. 버스 C를 사용하여 누산 명령만 누산기 레지스터의 전 48비트를 액세스할 수 있다.

재충전 비트가 설정되면, 레지스터는 사용 후에 공백으로 표시되고, 통상의 재충전 기구(ROB 상의 부분을 참조)으로 ROB로부터 재충전될 것이다. 재충전되기 전에 레지스터가 소스 오퍼랜드로서 다시 사용되지 않으면 파이프는 정지하지 않을 것이다. 재충된 데이터가 효력이 있기 전의 최소 사이클의 수(가장 좋은 경우-데이터는 ROB의 헤드에서 대기하고 있다)는 1 또는 2 중 하나일 것이다. 따라서, 재충전 요구를 따르는 명령에 재충전된 데이터를 사용하지 않는 것은 현명하다. 다음의 2개의 명령에 대하여 오퍼랜드의 사용을 피할 수 있다면, 이것은 보다 깊은 파이프라인 실행에 대한 성능손실을 방지할 것이다.

리파일 비트는 레지스터 수의 끝에 '^'을 붙임으로서 어셈블리에서 지정된다. 공백으로 표시된 레지스터의 부분은 레지스터 오퍼랜드에 의존한다. 각 레지스터의 2개의 절반은 재충전을 위해 독립하여 표시될 것이다(예컨대, X0.1^는 재충전을 위해 X0의 하부 절반만 표시할 것이고, X0^는 재충전을 위해 X0의 전체를 표시할 것이다). 48비트 레지스터의 상부 '절반'(비트 47:16)이 재충전되면, 데이터의 16비트는 비트 31:16으로 기록되고, 비트 47까지 부호 확장된다.

동일한 레지스터를 2회 재충전하는 것이 시도되면(예컨대, ADD X1, X0^, X0^), 한 번의 재충전만이 발생한다. 어셈블러는 오직 신택스 ADD X1, X0, X0^만을 허용해야 한다.

레지스터가 재충전되기 전에 레지스터 판독이 시도되면, 피클로는 레지스터가 재충전되는 것을 기다리지 않는다. 재충전을 위한 레지스터가 표시되고, 재충전된 값이 판독되기 전에 레지스터가 경신되면, 그 결과는 UNPREDICTABLE이다(예컨대 ADD X0, X0^, X1이 예언할 수 없는데, 그 이유는 ADD X0, X0^, X1이 재충전을 위해 X0을 표시한 후, X0 및 X1의 합을 ADD X0, X0^, X1 내에 배치함으로써 ADD X0, X0^, X1을 재충전하기 때문이다).

4비트 스케일 필드는 14개의 스케일 형태를 인코드한다.

-ASR #0, 1, 2, 3, 4, 6, 8, 10

-ASR #12-16

-LSL #1

병렬 Max/Min 명령은 한 개의 스케일을 제공하지 않기 때문에, 소스 2의 6비트의 일정한 변형이 이용되지 않는다(어셈블러에 의해 0으로 설정된다).

REPEAT 명령 내에서 레지스터 리맵핑이 지지되어, REPEAT가 루프를 회전시키지 않고 레지스터의 이동 '윈도우'에 액세스할 수 있다. 이것은 아래에 보다 상세히 설명되어 있다.

수신지 오퍼랜드는 다음의 7비트 형식을 갖는다.

	25	24	23	22	21	20	19
	F	SD	HL	DEST			

이 기본 인코딩에는 10번의 변형이 있다.

어셈블리 연상기호		25	24	23	22	21	20	19
Dx	1	0	1	0	Dx			
Dx^	2	1	1	0	Dx			
Dx.l	3	0	0	0	Dx			
Dx.l^	4	1	0	0	Dx			
Dx.h	5	0	0	1	Dx			
Dx.h^	6	1	0	1	Dx			
불확정		0	1	1	0000			
.l(레지스터는 되돌아가서 16비트를 기록하지 않는다.)	7	1	1	1	0	0	00	
^(레지스터는 되돌아가서 32비트를 기록하지 않는다.)	8	1	1	1	0	1	00	
.h^ (16-bit) 출력	9	1	1	1	1	0	00	
^ (32-bit) 출력	10	1	1	1	1	1	00	

레지스터 번호(Dx)는 16개의 레지스터 중 어느 것이 어드레스되는가를 나타낸다. Hi/Lo 비트와 사이즈 비트는 한 쌍의 16비트 레지스터로서 각 32비트 레지스터를 어드레스하는 것과 동시에 작용한다. 결과가 레지스터 뱅크 및/또는 출력 FIFO에 기록되는지의 여부에 관계없이, 이 사이즈 비트는 명령 형태로 정의한 바와 같이 적당한 플래그가 어떻게 설정되는가를 정의한다. 이것은 비교의 구성 및 비슷한 명령의 구성을 허용한다. 명령의 누산 클래스와의 덧셈은 그 결과를 레지스터로 되돌아가서 기록해야 한다.

다음의 표는 각 인코딩의 동작을 나타낸다.

인코딩	레지스터 기록	FIFO 기록	V FLAG
1	전 레지스터 기록	기록금지	32비트 오버플로
2	전 레지스터 기록	32비트 기록	32비트 오버플로
3	낮은 16비트를 Dx.l에 기록	기록금지	16비트 오버플로
4	낮은 16비트를 Dx.l에 기록	낮은 16비트 기록	16비트 오버플로
5	낮은 16비트를 Dx.h에 기록	기록금지	16비트 오버플로
6	낮은 16비트를 Dx.h에 기록	낮은 16비트 기록	16비트 오버플로
7	기록금지	기록금지	16비트 오버플로
8	기록금지	기록금지	32비트 오버플로
9	기록금지	낮은 16비트 기록	16비트 오버플로
10	기록금지	32비트 기록	32비트 오버플로

모든 경우에 있어서, 레지스터로 되돌아가 기록하거나 출력 FIFO 내로 삽입하기 전의 어떤 동작의 결과는 48비트의 양이다. 다음과 같은 2개의 경우가 있다.

기록이 16비트이면, 48비트 양은 하부 16비트[15:0]을 선택함으로써 16비트 양으로 감소된다. 이 명령이 포화되면, 그 값은 -2^15-2^15-1의 범위 내에서 포화될 것이다. 16비트 값은 지시된 레지스터로 되돌아가 기록되고, 기록 FIFO비트가 설정되면, 출력 FIFO로 되돌아가 기록된다. 16비트 값이 출력 FIFO에 기록되면, 그 값이 단일 32비트값으로서 출력 FIFO 내에 짝을 이루어 배치될 때 그것은 다음 16비트값이 기록될 때까지 보유된다.

32비트 기록에 대하여, 48비트 양이 하부 32비트[31:0]을 선택함으로써 32비트 양으로 감소된다.

32비트 및 48비트 기록에 대하여, 명령이 포화되면, 48비트값은 -2^31-1~2^31 범위에서 32비트로 변환될

것이다. 포화된 후에, 즉

- 누산기로 되돌아가서 기록되면, 전 48비트가 기록될 것이다.

- 32비트 레지스터로 되돌아가서 기록되면, 비트 [31:0]이 기록된다.

- 출력 FIFO로 되돌아가서 기록되면, 다시 비트[31:0]이 기록될 것이다.

수신지 사이즈는 레지스터 번호 후에 .s, .l 또는 .h에 의해 어셈블러에서 지정된다. 어떠한 레지스터도 되돌아가서 기록되지 않으면, 레지스터 번호는 중요하지 않으므로, 레지스터에 기록금지를 지시하기 위해 수신지 레지스터를 생략하거나, 또는 출력 FIFO에만 기록을 지시하기 위해 ^을 사용한다. 예컨대, SUB, X0, Y0은 CMP, X0, Y0 및 ADD^와 같고, X0, Y0은 출력 FIFO 내에 X0+Y0의 값을 배치한다.

한 개의 값에 대한 출력 FIFO에 어떠한 공간도 없으면, 피콜로는 공간이 이용가능하게 되는 것을 기다리지 않는다.

16비트 값이 예컨대, ADD X0.h^, X0, X2를 기록하면, 그 값은 두 번째의 16비트값이 기록될 때까지 래치된다. 2개의 값이 결합되어 32비트 번호로서 출력 FIFO 내에 배치된다. 기록된 첫 번째의 16비트값이 32비트 워드의 하부 절반에 나타난다. 출력 FIFO 내에 입력된 데이터는 16비트 데이터 또는 32비트 데이터 중 어느 하나로 표시되어, 엔디안이 큰 엔디안 시스템 상에서 수정될 수 있다.

32비트값이 2개의 16비트 기록 사이에서 기록되면, 그 동작은 규정되지 않는다.

REPEAT 명령 내에서 레지스터 리맵핑이 지지되어, REPEAT는 루프를 회전시키지 않고 레지스터의 이동 '윈도우'에 액세스할 수 있다. 이것은 아래에 보다 상세히 설명되어 있다.

본 발명의 바람직한 실시예에 있어서, REPEAT 명령은 레지스터 오퍼랜드가 한 개의 루프 내에 지정되는 방식을 변경하기 위해 하나의 기구를 제공한다. 이 기구하에, 액세스되어야 할 레지스터가 그 명령 내의 레지스터 오퍼랜드의 기능과, 레지스터 뱅크 내의 오프셋에 의해 결정된다. 이 오프셋이 프로그램가능한 방식으로, 바람직하게는 각 명령 루프의 끝부분에서 변경된다. 이 기구는 X, Y 및 Z 뱅크 내에 존재하는 레지스터와 관계없이 동작할 것이다. 바람직한 실시예에 있어서, 이 설비는 A 뱅크 내의 레지스터에 대하여는 이용할 수 없다.

논리적이고, 물리적인 레지스터의 개념이 사용될 수 있다. 명령 오퍼랜드는 논리적 레지스터 레퍼런스이고, 이들은 특정 피콜로 레지스터(10)를 식별하는 물리적 레지스터 레퍼런스에 맵핑된다. 재충전을 포함하는 모든 연산은 물리적 레지스터에 대하여 작용한다. 레지스터 리맵핑만이 피콜로 명령 스트림측에 발생하고, 피콜로 내에 로드된 데이터는 항상 물리적 레지스터로 정해져, 어떠한 리맵핑도 행해지지 않는다.

피콜로 코프로세서(4)의 다수의 내부 구성소자를 예시하는 블록도인 도 5를 참조하면서 리맵핑 기구에 대하여 설명할 것이다. 메모리로부터 ARM 코어(2)에 의해 검색된 데이터 항목은 발주 버퍼(12) 내에 배치되고, 피콜로 레지스터(10)는 도 2를 참조하여 이전에 설명한 방법으로 발주 버퍼(12)로부터 재충전된다. 캐시(6) 내에 저장된 피콜로 명령은 피콜로(4) 내의 명령 디코더(50)에 전달되고, 여기서 이들 피콜로 명령은 피콜로 프로세서 코어(54)에 전달되기 전에 디코드된다. 피콜로 프로세서 코어(54)는 곱셈/덧셈회로(20), 누적/누적처분회로(22) 및 도 3을 참조하면서 이전에 설명한 스케일/포화회로(24)를 포함한다.

명령 디코더(50)가 REPEAT 명령에 의해 식별된 명령 루프의 일부를 형성하는 명령들을 처리하고 있고, REPEAT 명령이 다수의 레지스터의 리맵핑이 발생해야 한다는 것을 지시했으면, 레지스터 래맵핑 논리(52)를 이용하여 필요한 리맵핑을 수행한다. 레지스터 리맵핑 논리(52)가 완전히 분리된 엔티티로서 명령 디코더(50)에 제공될 것이라는 것이 본 발명이 속하는 기술분야의 당업자에게서 분명해지더라도, 레지스터 래밉핑 논리(52)를 명령 디코더(50)의 일부라고 간주할 수 있다.

명령은 대체로 이 명령에 의해 요구된 데이터 항목을 포함하는 레지스터를 식별하는 1개 또는 그 이상의 오퍼랜드를 포함할 것이다. 예컨대, 대표적인 명령은 명령에 의해 요구된 데이터 항목을 포함하는 2개의 레지스터와 명령의 결과가 배치되어야 하는 어느 한 개의 레지스터를 식별하는, 2개의 소스 오퍼랜드와 1개의 수신지 오퍼랜드를 포함할 것이다. 레지스터 리맵핑 논리(52)는 명령 디코더(50)로부터 어느 한 개의 명령의 오퍼랜드를 수신하고, 이들 오퍼랜드는 논리적 레지스터 레퍼런스를 식별한다. 이 논리적 레지스터 레퍼런스에 근거하여, 레지스터 리맵핑 논리는 래밉핑이 적용되어야 하는지 아닌지의 여부를 결정할 것이고, 요구대로 물리적 레지스터 레퍼런스에 리맵핑을 적용할 것이다. 리맵핑을 적용해서는 안 된다고 결정하면, 논리 레지스터 레퍼런스가 물리 레지스터 레퍼런스로서 제공된다. 리맵핑을 행하는 바람직한 방법에 대해서는 후에 보다 상세히 설명할 것이다.

레지스터 리맵핑 논리로부터 출력된 각 물리적 레지스터 레퍼런스는 피콜로 코프로세서 코어(54)에 전달되어, 프로세서 코어가 물리적 레지스터 레퍼런스에 의해 식별된 특수 레지스터(10) 내의 데이터 항목에 명령을 적용할 수 있다.

바람직한 실시예의 리맵핑 기구에 의해 레지스터의 각 뱅크가 2개의 부분, 즉 레지스터가 리맵핑되는 부분과, 리맵핑없이 레지스터가 그들 본래의 레지스터 레퍼런스를 보유하는 부분으로 분리될 수 있다. 바람직한 실시예에 있어서, 이 리맵핑된 부분은 리맵핑되어 있는 레지스터 뱅크의 하부에서 시작한다.

리맵핑 기구는 다수의 파라미터를 사용하고, 레지스터 리맵핑 논리(52)가 다양한 파라미터를 사용한다는 것을 예시하는 블록도인 도 6을 참조하면서 이들 파라미터에 대해서 설명할 것이다. 이들 파라미터를 리맵핑되어 있는 뱅크 내의 한 점과 관련되어 있는 값에 주고, 이 점은 예컨대, 뱅크의 하부이다.

레지스터 리맵핑 논리(52)는 2개의 주 논리블록, 즉 리맵 블록(56) 및 베이스 갱신 블록(58)을 구비한다고 간주될 수 있다. 레지스터 리맵핑 논리(52)는 논리 레지스터 레퍼런스에 더해져야 할 오프셋값을 제공하는 베이스 포인터를 사용하고, 이 베이스 포인터 값은 베이스 갱신 블록(58)에 의해 리맵 블록(56)에

제공된다.

BASESTART 신호는 일부 다른 값이 지정되더라도, 베이스 포인터의 초기값을 규정하기 위해 사용될 수 있는데, 이것은 일반적으로 예컨대 0이다. 이 BASESTART 신호는 베이스 갱신 블록(58) 내의 멀티플렉서(60)에 전달된다. 명령 루프의 첫 번째의 반복시, BASESTART 신호는 멀티플렉서에 의해 기억소자(66)에 전달되지만, 다음의 이 루프의 반복에 대하여는, 다음 베이스 포인터 값이 멀티플렉서(60)에 의해 기억소자(66)에 공급된다.

기억소자(66)의 출력은 현 베이스 포인터 값으로서 리맵 논리(56)에 전달되고, 베이스 갱신 논리(58) 내의 가산기(62)의 입력 중 하나에 전달된다. 가산기(62)는 또한 베이스 증분값을 제공하는 BASEINC 신호를 수신한다. 이 가산기(62)는 기억소자(66)에 의해 공급된 현 베이스 포인터 값을 BASEINC값만큼 증가시키기 위해 배치되고, 그 결과를 모듈로회로(64)에 전달하기 위해 배치된다.

모듈로회로는 또한 BASEWRAP 값을 수신하고, 이 값을 가산기(62)로부터의 출력 베이스 포인터 신호와 비교한다. 증분된 베이스 포인터 값이 BASEWRAP 값과 같거나 또는 BASEWRAP 값을 초과하면, 신규 베이스 포인터는 신규 오프셋 값와 겹쳐 있다. 모듈로회로(64)의 출력은 기억소자(66)에 기억되어야 할 다음 베이스 포인터 값이다. 이 출력은 멀티플렉서(60)에 제공되고, 멀티플렉서(60)로부터 기억소자(66)에 제공된다.

그러나, 이 다음 베이스 포인터 값은 BASEUPDATE 신호가 REPEAT 명령을 처리하는 루프 하드웨어로부터 기억소자(66)에 의해 수신될 때까지 기억소자(66) 내에 기억될 수 없다. BASEUPDATE 신호는 루프 하드웨어에 의해 주기적으로, 예컨대 명령 루프가 반복될 때마다 생성될 것이다. BASEUPDATE 신호가 기억소자(66)에 의해 수신되면, 기억소자는 멀티플렉서(60)에 의해 제공된 다음 베이스 포인터 값과 이전 베이스 포인터 값을 겹쳐 쓸 것이다. 이와 같이, ReMap 논리(58)에 제공된 베이스 포인터 값은 신규 베이스 포인터 값으로 변경될 것이다.

레지스터 뱅크의 리맵핑된 부분 내의 액세스되어야 할 물리적 레지스터는 명령의 오퍼랜드 내에 포함된 논리적 레지스터 레퍼런스와, 베이스 갱신 논리(58)에 의해 제공된 베이스 포인터 값의 덧셈에 의해 결정된다. 이 덧셈은 덧셈기에 의해 행해지고, 출력은 모듈로회로(70)에 전달된다. 바람직한 실시예에 있어서, 모듈로회로(70)는 또한 레지스터 중첩 값을 수신하고, 가산기(68)로부터의 출력신호(논리적 레지스터 레퍼런스와 베이스 포인터 값의 덧셈)가 레지스터 중첩 값을 초과하면, 그 결과는 리맵핑된 영역의 하부까지 겹칠 것이다. 모듈로회로(70)의 출력은 멀티플렉서(72)에 제공된다.

REGCOUNT 값은 리맵핑되어야 할 뱅크 내의 레지스터의 수를 식별하는 리맵(Remap) 블록(56) 내의 논리(74)에 제공된다. 논리(74)는 논리적 레지스터 레퍼런스와 이 REGCOUNT 값을 비교하고, 그 비교결과에 의존하여 제어신호를 멀티플렉서(72)에 전달한다. 멀티플렉서(72)는 그것의 2개의 입력으로서 논리 레지스터 레퍼런스와 모듈로회로(70)로부터의 출력(리맵핑된 레지스터 레퍼런스)을 수신한다. 본 발명의 바람직한 실시예에 있어서, 논리 레지스터 레퍼런스가 REGCOUNT 값보다 작으면, 논리(74)는 멀티플렉서에게 물리적 레지스터 레퍼런스로서 리맵핑된 레지스터 레퍼런스를 출력하라고 지시한다. 그러나, 논리 레지스터 레퍼런스가 REGCOUNT 값보다 크거나 또는 REGCOUNT 값과 같으면, 논리(74)는 멀티플렉서(72)에게 물리적 레지스터 레퍼런스로서 논리적 레지스터 레퍼런스를 직접 출력하라고 지시한다.

이전에 설명한 바와 같이, 바람직한 실시예에 있어서, 이것은 리랩핑 기구에 호소하는 REPEAT 명령이다. 후에 보다 상세히 설명한 바와 같이, REPEAT 명령은 하드웨어에 4개의 0 사이클 루프를 제공한다. 이들 하드웨어 루프에 대해서는 명령 디코더(50)의 일부로서 도 5에 예시되어 있다. 명령 디코더(50)가 캐시(6)로부터 명령을 요구할 때마다, 캐시는 그 명령을 명령 디코더에 반환하고, 그래서 명령 디코더는 반환된 명령이 REPEAT 명령인지의 여부를 결정한다. 만일 그렇다면, 하드웨어 루프 중 하나를 구성하여 그 REPEAT 명령을 처리한다.

각 반복명령은 루프에서의 명령 수 및 횟수를 지정하여 (상수 또는 피콜로 레지스터로부터의 판독 중 어느 하나인) 루프를 순회한다. 2개의 연산코드 REPEAT 및 NEXT는 하드웨어 루프를 정의하기 위해 제공되고, NEXT 연산코드는 단순히 구분문자로서 사용되고, 어느 하나의 명령으로서 어셈블리되지 않는다. REPEAT는 루프의 처음에 놓이고, NEXT는 루프의 끝부분의 범위를 정하기 때문에, 어셈블러는 루프 본체에서 명령의 수를 산출할 수 있다. 바람직한 실시예에 있어서, REPEAT 명령은 레지스터 리맵핑 논리(52)가 사용해야 할 REGCOUNT, BASEINC, BASEWRAP 및 REGWRAP 파라미터 등의 리맵핑 파라미터를 포함한다.

다수의 레지스터는 레지스터 리맵핑 논리가 사용한 리맵핑 파라미터를 기억하기 위해 제공될 수 있다. 이들 레지스터 내에서, 다수의 소정의 리맵핑 파라미터의 세트가 제공될 수 있지만, 사용자가 규정된 리맵핑 파라미터의 기억을 위해 일부 레지스터를 남겨 둔다. REPEAT 명령으로 지정된 리맵핑 파라미터가 소정의 리맵핑 파라미터의 세트 중 하나와 같으면, 적절한 REPEAT 인코딩이 사용되고, 이 인코딩에 의해 멀티플렉서 등은 이 레지스터로부터 직접 레지스터 리맵핑 논리로 적절한 리맵핑 파라미터를 제공한다. 한편, 리맵핑 파라미터가 소정의 리맵핑 파라미터의 세트 중 어느 것인가와 동일하지 않으면, 어셈블러는 사용자가 규정된 레지스터 리맵핑 파라미터의 구성을 허용하는 리맵핑 파라미터 이동 명령(Remapping Parameter Move Instruction: RMOV)을 발생할 것이고, 이 RMOV 명령은 REPEAT 명령 전에 온다. 바람직하게는, 사용자가 규정된 리맵핑 파라미터는 그러한 사용자가 규정된 리맵핑 파라미터를 기억하기 위해 옆에 둔 레지스터 내에 RMOV 명령에 의해 배치될 것이고, 멀티플렉서는 이들 레지스터의 내용을 레지스터 리맵핑 논리에 전달하도록 프로그램될 것이다.

바람직한 실시예에 있어서, REGCOUNT, BASEINC, BASEWARP 및 REGWRAP 파라미터는 아래의 챠트에서 식별된 값 중 하나를 갖는다.

파라미터	설명
REGCOUNT	이것은 16비트 레지스터의 수를 식별하여 리맵핑을 행하고, 값 0, 2, 4, 8을 갖는다. REGCOUNT 이하의 레지스터가 리맵핑되고, REGCOUNT 이상 또는 REGCOUNT와 같은 레지스터는 직접 액세스된다.
BASEINC	이것은 얼마나 많은 16비트 레지스터에 의해 베이스 포인터가 각 루프의 반복의 끝부분에서 증가되는가를 정의한다. 바람직한 실시예에 있어서는, 사실상 소망한다면 적당한 곳의 음의 값을 포함하는 다른 값을 갖더라도, 값 1, 2 또는 4를 갖는다.
BASEWRAP	이것은 베이스 계산의 상한을 결정한다. 베이스 중첩 계수는 값 2, 4, 8을 갖는다.
REGWRAP	이것은 리맵 계산의 상한을 결정한다. 레지스터 중첩 계수는 값 2, 4, 8을 갖는다. REGWRAP는 REGCOUNT와 같도록 선택될 것이다.

도 6으로 되돌아가서, 다양한 파라미터가 리맵 블록 56에 의해 어떻게 사용되는가의 예는 다음과 같다(이 예에 있어서, 논리적, 물리적 레지스터 값은 특수한 뱅크와 관련되어 있다).

```
if(Logical Register < REGCOUNT)
Physical Register = (Logical Register + Base)MOD REGCOUNT
else
Physical Register = Logical Register
end if
```

루프의 끝부분에서, 다음의 루프의 반복이 시작하기 전에, 다음의 베이스 포인터에 대한 갱신이 베이스 갱신 논리(58)에 의해 행해진다.

```
Base = (Base + BASEINC) MOD BASEWRAP
```

리맵핑 루프의 끝부분에서, 레지스터 리맵핑이 전환될 것이고, 그후 모든 레지스터는 물리적 레지스터로서 액세스될 것이다. 바람직한 실시예에 있어서, 단 한 개의 리맵핑 REPEAT만이 언제라도 활성화될 것이다. 루프는 여전히 네스트(nest)될 수 있지만, 한 개만 어떤 특정 시간에 리맵핑 변수를 갱신할 수 있다. 그러나, 소망한다면, 리맵핑 반복이 네스트될 수 있다는 것은 인식될 것이다.

본 발명의 바람직한 실시예에 따른리맵핑 기구를 사용한 결과로서 코드 밀도에 관하여 획득된 이점을 예시하기 위해, 대표적인 블록 필터 알고리즘에 관하여 설명할 것이다. 우선, 블록 필터 알고리즘의 원리에 대하여 도 7을 참조하면서 설명할 것이다. 도 7에 나타낸 바와 같이, 누산기 레지스터 A0는 다수의 곱셈 동작의 결과를 누적하기 위해 배치되고, 이 곱셈동작은 데이터 항목 d0과 계수 c0의 곱셈, 데이터 항목 d1과 계수 c1의 곱셈, 데이터 항목 d2와 계수 c2의 곱셈 등이다. 레지스터 A1은 비슷한 곱셈 동작의 세트의 결과를 누적하지만, 이 때 계수의 세트가 시프트되어, c0은 d1과 곱해지고, c1은 d2와 곱해지며, c2는 d3과 곱해진다. 비슷하게, 레지스터 A2는 또 다른 단계를 우측으로 시프트한 계수 값과 데이터 값을 곱셈한 결과를 누적하기 때문에, co는 d2와 곱해지고, c1은 d3과 곱해지면, c2는 d4와 곱해진다. 이 시프트, 곱셈, 누적과정을 반복하여 그 결과를 레지스터 A30에 배치한다.

본 발명의 바람직한 실시예에 따른 레지스터 리맵핑이 사용되지 않으면, 블록 필터 명령을 수행하도록 다음의 명령 루프를 요구할 것이다.

```
; 4개의 신규 데이터 값으로 시작
ZERO {A0-A3}                          ; 누산기를 제로로 한다.

REPEAT  Z1                                    ; Z1= (계수의 수/4)
; 처음에 다음 4개의 계수를 처리:
; a0 += d0*c0+d1*c1+d2*c2+d3*c3
; a1 += d1*c0+d2*c1+d3*c2+d4*c3
; a2 += d2*c0+d3*c1+d4*c2+d5*c3
; a3 += d3*c0+d4*c1+d5*c2+d6*c3
```

```
MULA        A0, X0.l^, Y0.l, A0        ; a0 += d0*c0, 및 d4 로드
MULA        A1, X0.h, Y0.l, A1         ; a1 += d1*c0
MULA        A2, X1.l, Y0.l, A2         ; a2 += d2*c0
MULA        A3, X1.h, Y0.l^, A3        ; a3 += d3*c0, 및 c4 로드
MULA        A0, X0.h^, Y0.h, A0        ; a0 += d1*c1 및 d5 로드
MULA        A1, X1.l, Y0.h, A1         ; a1 += d2*c1
MULA        A2, X1.h, Y0.h, A2         ; a2 += d3*c1
MULA        A3, X0.l, Y0.h^, A3        ; a3 += d4*c1, 및 c5 로드
MULA        A0, X1.l^, Y1.l, A0        ; a0 += d2*c2 및 d6 로드
MULA        A1, X1.h, Y1.l, A1         ; a1 += d3*c2
MULA        A2, X0.l, Y1.l, A2         ; a2 += d4*c2
MULA        A3, X0.h, Y1.l^, A3        ; a3 += d5*c2, 및 c6 로드
MULA        A0, X1.h^, Y1.h, A0        ; a0 += d3*c3 및 d7 로드
MULA        A1, X0.l, Y1.h, A1         ; a1 += d4*c3
MULA        A2, X0.h, Y1.h, A2         ; a2 += d5*c3
MULA        A3, X1.l, Y1.h^, A3        ; a3 += d6*c3 및 c7 로드
NEXT
```

이 예에서, 데이터 값은 레지스터의 X 뱅크 내에 배치되고, 계수 값은 레지스터의 Y 뱅크 내에 배치된다. 첫 번째의 단계로서, 4개의 누산기 레지스터 A0, A1, A2 및 A3은 0으로 설정된다. 일단 누산기 레지스터가 리세트되면, 명령 루프가 입력되고, REPEAT 및 NEXT 명령에 의해 경계가 정해진다. 값 Z1은 명령 루프가 반복되어야 하는 횟수를 식별하고, 후에 설명되는 이유로, 이것은 실제로 계수(c0, c1, c2 등)/4와 동일할 것이다.

명령 루프는 16개의 곱셈 누산 명령(MULA)을 구비하는데, 이 곱셈 누산 명령은 그 루프를 통해서 처음으로 반복한 후에 REPEAT와 제 1 MULA 명령 사이의 상기 코드에 나타난 계산 결과를 포함하는 레지스터 A0, A1, A2, A3로 된다. 곱셈 누산 명령이 어떻게 동작하는가를 예시하기 위해, 첫 번째의 4개의 MULA 명령에 대하여 고려할 것이다. 제 1 명령은 X 뱅크 레지스터 제로의 첫 번째 또는 하부 16비트 내부의 데이터 값과 Y 뱅크 레지스터 제로 내부의 하부 16비트를 곱하고, 그 결과를 누산기 레지스터 A0과 더한다. 동시에, X 뱅크 레지스터 제로의 하부 16비트는 재충전 비트로 표시되고, 이것은 레지스터의 일부가 신규 데이터 값으로 재충전될 수 있다는 것을 표시한다. 도 7로부터 명백한 바와 같이, 일단 데이터 항목 d0이 계수 c0과 곱해졌기 때문에(이것은 제 1 MULA 명령으로 표시되어 있다), d0은 더 이상 블록 필터 명령의 나머지에 대하여 요구되지 않으므로, 신규 데이터 값으로 대체될 수 있다.

제 2 MULA 명령은 X 뱅크 레지스터 제로의 두 번째 또는 상부 16비트와 Y 뱅크 레지스터 제로의 하부 16비트를 곱한다(이것은 도 7에 나타낸 $d1 \times c0$을 나타낸다). 비슷하게, 제 3 및 제 4 MULA 명령은 곱셈 $d2 \times c0$, $d3 \times c0$을 각각 나타낸다. 도 7에 나타낸 바와 같이, 일단 이들 4개의 계산이 행해졌으면, 계수 C0은 더 이상 요구되지 않으므로, 레지스터 Y0.l은 재충전 비트로 표시되어, 또 다른 계수(c4)와 겹쳐서 기록될 수 있다.

다음 4개의 MULA 명령은 계산 $d1 \times c1$, $d2 \times c1$, $d3 \times c1$, $d4 \times c1$을 각각 나타낸다. 일단 계산 $d1 \times c1$이 행해졌으면, d1이 더 이상 요구되지 않기 때문에 레지스터 X0.h는 재충전 비트로 표시된다. 비슷하게, 일단 모든 4개의 계산이 행해졌으면, 계수 c1은 더 이상 요구되지 않기 때문에, 레지스터 Y0.h는 재충전을 위해 표시된다. 비슷하게, 다음 4개의 MULA 명령은 계산 $d2 \times c2$, $d3 \times c2$, $d4 \times c2$, $d5 \times c2$0에 대응하고, 마지막 4개의 계산은 계산 $d3 \times c3$, $d4 \times c3$, $d5 \times c3$, $d6 \times c3$0에 대응한다.

상술한 실시예에 있어서, 레지스터가 리맵할 수 없기 때문에, 각 곱셈동작은 요구된 특정 레지스터가 오퍼랜드 내에 지정되어 있는 상태로 명백하게 재생되어야 한다. 일단 16개의 MULA 명령이 행해졌으면, 명령 루프는 계수 c4~c7 및 데이터 항목 d4~d10에 대하여 반복될 수 있다. 또한, 이 루프가 반복마다 4개의 계수 값에 작용하기 때문에, 계수 값의 수는 4의 배수이어야 하고, 계산 Z1 = 계수의 수/4가 산출되어야 한다.

본 발명의 바람직한 실시예에 따른 리맵핑 메카니즘을 사용함으로써, 명령 루프는 극적으로 감소될 수 있어, 다른 방법으로 요구되었던 16개의 곱셈 누산 명령보다는 오히려 4개의 곱셈 누산 명령을 포함한다. 리맵핑 기구를 사용하면, 코드는 다음과 같이 기록될 수 있다.

```
; 4개의 신규 데이터 값으로 시작
제로 {A0-A3}                          ; 누산기를 제로로 한다.
Z1, X++ n4 w4 r4, Y++ n4 w4 r4 반복;  Z1=(계수의 수)
; 리맵핑은 X 및 Y 뱅크에 적용된다.
```

```
; 이들 뱅크 내의 4개의 16비트 레지스터가 리맵핑된다.
; 양쪽 뱅크의 베이스 포인트는 각각에 대하여 1개씩 증가된다.
; 루프의 반복
; 베이스 포인터가 뱅크 내의 4개의 레지스터에 도달할 때,
; 베이스 포인터가 증첩한다.
MULA        A0, X0.l^, Y0.l, A0        ; a0 += d0*c0, 및 d4 로드
MULA        A1, X0.h, Y0.l, A1         ; a1 += d1*c0
MULA        A2, X1.l,  Y0.l, A2        ; a2 += d2*co
MULA        A3, X.1.h, Y0.l^, A3       ; a3 += d3*c0, 및 c4 로드
NEXT                                   ; 루프를 순회하고, 리맵핑을
                                       진행한다.
```

전과 같이, 제 1 단계는 4개의 누산기 레지스터 A0-A3을 제로로 설정할 것이다. 그 후, 명령 루프가 입력되고, REPEAT 및 NEXT 연산코드에 의해 경계가 정해진다. 이 REPEAT 명령은 그것과 관련된 다수의 파라미터를 갖고, 이들 파라미터는 다음과 같다.

X++ : BASEINC는 레지스터의 X 뱅크에 대하여 '1'이라는 것을 나타낸다.

n4 : REGCOUNT가 '4'이고, 따라서 첫 번째의 4개의 X 뱅크 레지스터 X0.l ~X1.h가 리맵핑된다는 것을 나타낸다.

w4 : BASEWRAP이 레지스터의 X 뱅크에 대하여 '4'라는 것을 나타낸다.

r4 : REGWRAP이 레지스터의 X 뱅크에 대하여 '4'라는 것을 나타낸다.

Y++ : BASEINC이 레지스터의 Y 뱅크에 대하여 '1'이라는 것을 나타낸다.

n4 : REGCOUNT가 '4'이고, 따라서 첫 번째의 4개의 Y 뱅크 레지스터 Y0.l ~Y1.h가 리맵핑된다는 것을 나타낸다.

w4 : BASEWRAP가 레지스터의 Y 뱅크에 대하여 '4'라는 것을 나타낸다.

r4 : REGWRAP가 레지스터의 Y 뱅크에 대하여 '4'라는 것을 나타낸다.

값 Z1이 종래의 예에서와 같이, 계수의 수/4와 같기보다는 오히려 계수의 수와 같다는 것에 주의해야 한다.

명령 루프의 첫 번째의 반복에 대하여, 베이스 포인터 값은 0이므로, 어떠한 리맵핑도 없다. 그러나, 다음의 루프 실행시, 베이스 포인터 값은 X 및 Y 뱅크에 대하여 '1'이므로, 오퍼랜드는 다음과 같이 맵핑될 것이다.

X0.l은 X0.h로 된다.

X0.h는 X1.l로 된다.

X1.l은 X1.h로 된다.

X1.h는 X0.l로 된다(BASEWRAP가 '4'이기 때문에).

Y0.l은 Y0.h로 된다.

Y0.h는 Y1.l로 된다.

Y1.l은 Y1.h로 된다.

Y1.h는 Y0.l로 된다(BASEWRAP가 '4'이기 때문에).

따라서, 제 2 반복에 대하여, 4개의 MULA 명령은 실제로 본 발명의 리맵핑을 포함하지 않는 이전에 설명한 예에서 제 5 내지 제 8 MULA 명령으로 표시된 계산을 행한다. 비슷하게, 루프를 통한 제 3 및 제 4 반복은 종래의 코드의 제 9 내지 제 12, 및 제 13 내지 제 16 MULA 명령에 의해 전에 행해진 계산을 행한다.

따라서, 종래에 요구된 16개보다는 오히려 4개의 명령만이 제공되어야 하기 때문에, 상기 코드가 종래의 코드와 동일한 블록 필터 알고리즘을 정확히 수행하지만, 4개의 인자에 의해 루프 본체 내의 코드 밀도를 향상시킨다는 것을 알 수 있다.

본 발명의 바람직한 실시예에 따른 레지스터 리맵핑 기술을 사용함으로써, 다음의 이점이 실현될 수 있다.

1. 이것은 코드 밀도를 향상시킨다.

2. 이것은 일정한 상황에서 레지스터가 공백이라는 것을 피룔로 레코더 버퍼에 의해 재충전되어 있는 레지스터에 표시하는 것으로부터 잠복시간을 숨길 수 있다. 이것은 증가된 코드 사이즈의 비용으로 루프를 순회하지 않고 달성될 수 있다.

3. 이것에 의해 레지스터의 변수가 액세스될 수 있고, 행해진 루프 반복의 수를 변경함으로써 액세스된 레지스터의 수가 변경될 수도 있다.

4. 이것은 알고리즘 개발을 용이하게 할 수 있다. 적당한 알고리즘에 대하여, 프로그래머는 알고리즘의 n번째의 단계에 대한 코드의 일부를 생성한 후, 레지스터 리맵핑을 이용하여 그 방식을 데이터의 불확실한 세트에 적용할 수 있다.

본 발명의 범위로부터 벗어나지 않고 상술한 레지스터 리맵핑 메카니즘에 대하여 어떤 변경을 행할 수 있다는 것은 분명할 것이다. 예컨대, 레지스터 10의 뱅크는 명령 오퍼랜드에서 프로그래머에 의해 지정될 수 있는 것보다 더 물리적 레지스터를 제공할 수 있다. 이들 여분의 레지스터는 직접 액세스될 수 없지만, 레지스터 리맵핑 메카니즘은 이들 레지스터를 이용가능하게 할 수 있다. 예컨대, 레지스터의 X 뱅크가 프로그래머에 이용할 수 있는 4개의 32비트 레지스터를 갖고, 따라서 8개의 16비트 레지스터가 물리적 레지스터 레퍼런스에 의해 지정될 수 있는 전술한 예를 고려한다. 레지스터의 X 뱅크는 실제로 예컨대 6개의 32비트 레지스터로 구성될 수 있고, 이 경우에, 프로그래머에 직접 액세스할 수 없는 4개의 부가적인 16비트 레지스터가 있을 것이다. 그러나, 이들 여분의 4개의 레지스터는 리맵핑 메카니즘에 의해 이용가능하게 될 수 있고, 그것에 의해 데이터 항목의 저장을 위해 부가적인 레지스터를 제공할 수 있다.

다음의 어셈블러 신택스가 사용될 것이다.

>>는 시프트 오퍼랜드가 부(-)인 경우의 논리적인 우측 시프트 또는 좌측 시프트를 의미한다(아래의 <lscale>참조).

->>는 시프트 오퍼랜드가 부(-)인 경우의 산술적인 우측 시프트 또는 좌측 시프트를 의미한다(아래의 <scale>참조).

ROR은 우측 회전(Rotate Right)을 의미한다.

SAT(a)는 a의 포화된 값을 의미한다(수신지 레지스터의 사이즈에 의존 하는 16 또는 32비트로 포화된 값). 특히, 16비트로 포화시키기 위해, +0x7fff 보 다 큰 어떤 값은 +0x7fff로 대체되고, -0x8000보다 작은 어떤 값은 -0x8000 으로 대체된다. 32비트로의 포화는 최종의 +0x7fffffff 및 -0x80000000과 비 슷하다. 수신지 레지스터가 48비트이면, 여전히 32비트에서 포화된다.

소스 오퍼랜드 1은 다음의 형식 중 하나일 것이다.

<src1>은 [Rn:Rn.l:Rn.h:Rn.x][^]에 대하여 쇼트핸드(shorthand)이다. 즉, 7비트의 소스 지정자(specifier)가 모두 유효하고, 레지스터는 (임의로 교환된) 32비트값 또는 부호 확장된 16비트값으로서 판독된다. 누산기에 대하여는 하부 32비트만이 판독된다. ^는 레지스터 재충전을 나타낸다.

<src1_16>는 [Rn.l:Rn.h][^]의 단축형이다. 16비트값만이 판독될 수 있다.

<src1_32>는 [Rn:Rn.x][^]의 단축형이다. 32비트값만이 임의로 교환된 상부 및 하부 절반으로 판독될 수 있다.

소스 오퍼랜드 2는 다음의 형식 중 하나일 것이다.

<src2>는 아래의 3개의 옵션에 대한 쇼트핸드일 것이다.

_ 최종 결과의 스케일(<scale>)을 더한, [Rn:Rn.l:Rn.h:Rn.x][^] 형태 의 소스 레지스터.

_ 최종 결과의 스케일이 없는, 임의로 시프트된 8 비트 상수(<immed_8>).

- 최종 결과의 스케일(<scale>)을 더한, 6 비트 상수 (<immed_6>).

<src2_maxmin>은 <src2>와 동일하지만, 스케일은 행해지지 않는다.

<src2_shift> 시프트 명령은 한정된 <src2>의 서브세트를 제공한다. 자세한 것은 상기 참조.

<src2_par>는 <src2_shift>에 관한 한

제 3 오퍼랜드를 지정하는 명령에 대하여:

<acc>는 4개의 누산기 레지스터[A0:A1:A2:A3] 중 어느 것인가의 약어이다.

48비트가 모두 판독된다. 어떠한 재충전도 지정될 수 없다.
수신지 레지스터는 다음과 같은 형식을 갖는다.
<dest>는 [Rn:Rn.l:Rn.h:.l:][^]의 단축형이다. '.' 확장도 없이, 전 레지스터가 기록된다(누산기의 경우에는 48비트). 레지스터로 되돌아가는 어떠한 기록도 요구되지 않는 경우에, 사용된 레지스터는 중요하지 않다. 어셈블러는 되돌아가서 기록하는 것이 요구되지 않는다는 것을 나타내기 위한 수신지 레지스터, 또는 되돌아가서 기록하는 것이 요구되지 않지만 결과가 16비트 양이더라도 플래그가 설정되어야 한다는 것을 나타내기 위한 '.l'을 생략한다. ^는 출력 FIFO에 그 값이 기록된다는 것을 나타낸다.
<scale>는 다수의 산술 스케일을 나타낸다. 14개의 이용가능한 스케일이 있다.
ASR #0, 1, 2, 3, 4, 6, 8, 10
ASR #12~16
LSS #1
<immed_8>은 부호없는 8비트 즉치를 나타낸다. 이것은 0, 8, 16 또는 24의 시프트로 좌측으로 순환된 바이트로 구성된다. 따라서, 값 0xYZ000000, 0x00YZ0000, 0x0000YZ00 및 0x000000YZ는 어떤 YZ에 대하여 인코드될 수 있다. 이 순환은 2비트 양으로서 인코드된다.
<imm_6>은 부호없는 6비트 즉치를 나타낸다.
<PARAMS>는 레지스터 리맵핑을 지정하기 위해 사용되고, 다음의 형식을 갖는다.
<BANK><BASEINC>n<RENUMBER>w<BASEWRAP>
<BANK>는 [X:Y:Z]일 수가 있다.
<BASEINC>는 [++:+1:+2:+4]일 수가 있다.
<RENUMBER>는 [0:2:4:8]일 수가 있다.
<BASEWRAP>는 [2:4:8]일 수가 있다.

표현 <cond>는 다음의 조건 코드 중 어느 하나에 대한 쇼트핸드이다. 부호없는 LS 및 HI 코드는 보다 유용한 부호 오버플로/언더플로 테스트로 대체되었기 때문에 인코딩은 ARM과는 조금 다르다는 것에 유념해라. V 및 N 플래그는 ARM과는 다르게 피콜로에 대하여 설정되어, 조건 테스트에서 플래그 테스트로의 변환은 ARM과 동일하지 않는다.

0000	EQ	Z=0	마지막 결과는 0.
0001	NE	Z=1	마지막 결과는 0이 아님.
0010	CS	C=1	시프트/MAX 동작 후에 사용.
0011	CC	C=0	
0100	MI/LT	N=1	최종 결과는 부(-).
0101	PL/GE	N=0	최종 결과는 정(+).
0110	VS	V=1	부호 오버플로/최종결과에 대한 포화.
0111	VC	V=0	오버플로 없음/ 최종결과에 대한 포화.
1000	VP	V=1 & N=0	최종결과에 대한 정(+)의 오버플로.
1001	VN	V=1 & N=1	최종결과에 대한 부(-)의 오버플로.
1010	보류		
1011	보류		
1100	GT	N=0 & Z=0	
1101	LE	N=1 : Z=1	
1110	AL		
1111	보류		

피콜로는 부호 양을 처리하기 때문에, 부호없는 LS 및 HI 조건은 어떤 오버플로의 방향을 설명하는 VP 및 VN으로 대체되었다. ALU의 결과는 48비트이기 때문에, MI 및 LT는 PL 및 GE와 비슷하게 같은 기능을 수행한다. 이것은 미결정의 확장을 위해 3개의 슬롯을 남겨 둔다.

만약 다른 방법으로 표시하지 않았다면, 모든 동작에는 부호가 있다.

첫 번째와 두 번째의 조건코드는 각각 다음의 것으로 구성된다.

N - 부(-)

Z - 제로

C - 캐리/부호없는 오버플로

V - 부호 오버플로

연산명령은 2개의 형태, 즉 병렬 및 '전 폭(full width)'으로 분리될 수 있다. '전 폭' 명령만이 첫 번째의 플래그를 설정하고, 병렬동작은 결과의 상부 및 하부 16비트 절반에 근거하여 첫 번째와 두 번째의 플래그를 설정한다.

N, Z 및 V 플래그는 스케일이 수신지에 기록하기 전을 제외하고 적용된 후에 전 ALU 결과에 근거하여 계산된다. ASR은 항상 그 결과를 저장하기 위해 요구된 비트의 수를 줄일 것이고, 그러나 ASL은 그것을 증가시킬 것이다. 이 피콜로 절단을 피하기 위해, ASL 스케일이 적용될 때, 비트의 수를 제한하도록 48비트 결과에 대하여 0검출 및 오버플로를 수행해야 한다.

N 플래그가 산출되어 추정 부호 산술이 행해지고 있다. 이것은 오버플로가 발생할 때, 입력 오퍼랜드에 부호가 붙는지 아닌지에 의존하여, 그 결과의 최상위 비트가 C 플래그 또는 N 플래그 중 어느 하나이기 때문이다.

V 플래그는 정확도의 손실이 있으면 그 손실이 결과를 선택된 수신지에 기록한 결과로서 발생한다는 것을 나타낸다. 어떠한 되돌아가서 기록하는 것이 선택되지 않으면, '사이즈'는 여전히 수반되고, 오버플로 플래그는 정확하게 설정된다. 오버플로는,

- 결과가 -2^15~2^15-1의 범위에 있지 않을 때, 16비트 레지스터에 기록하는 경우.

- 결과가 -2^31~2^31-1의 범위에 있지 않을 때, 32비트 레지스터에 기록하는 경우에 발생할 수 있다.

병렬 덧셈/뺄셈 명령은 결과의 상부 및 하부 절반에 독립하여 N, Z 및 V 플래그를 설정한다.

누산기에 기록할 때, V 플래그는 마치 32비트 레지스터에 기록하는 것처럼 설정된다. 이것은 32비트 레지스터로서 누산기를 사용하도록 포화 명령을 허용하는 것이다.

포화 절대 명령(SABS)은 또한 입력 오퍼랜드의 절대값이 지정된 수신지에 맞으면 오버플로 플래그를 설정한다.

캐리 플래그는 덧셈 및 뺄셈 명령에 의해 설정되고, MAX/MIN, SABS 및 CLB 명령에 의해 '바이너리' 플래그로서 사용된다. 곱셈 연산을 포함하는 모든 다른 명령은 캐리 플래그를 보존한다.

덧셈 및 뺄셈 동작에 대하여, 캐리는 수신지가 32비트 또는 16비트인지의 여부에 근거하여, 비트 31 또는 비트 15 중 어느 하나 또는 결과에 의해 발생되는 것이다.

표준 연산명령은 플래그 설정방법에 따라, 수의 형태로 분리될 수 있다.

덧셈 및 뺄셈 명령의 경우에 있어서, N 비트가 설정되면, 모든 플래그가 보존된다. N비트가 설정되지 않으면, 플래그는 다음과 같이 갱신된다.

Z는 전 48비트 결과가 0인 경우에 설정된다.

N은 전 48비트 결과가 비트 47세트를 가졌을 때 설정된다(부(-)).

V는 수신지 레지스터가 16비트이고, 부호 결과가 16비트 레지스터 내에 맞지 않는 경우(-2^15<=x<2^15의 범위에 없는 경우), 또는 수신지 레지스터가 32/48 비트 레지스터이고, 부호 결과가 32비트 내에 맞지 않는 경우에 설정된다.

<dest>가 32 또는 48 비트 레지스터인 경우에, C 플래그는 <src1>와 <src2>를 합계할 때 비트 31의 캐리가 있으면 또는 <src1>로부터 <src2>를 뺄셈할 때 비트 31로부터 어떠한 빌림도 발생하지 않으면 설정된다(ARM에 대하여 동일한 캐리 값을 기대할 것이다). <dest>가 16비트 레지스터이면, C 플래그는 합계의 비트 15의 캐리가 있으면 설정된다.

두 번째의 플래그(SZ, SN, SV, SC)가 보존된다.

이들 명령의 경우에는, 48비트 레지스터로부터 곱셈 또는 누산을 수행한다.

Z는 전 48비트 결과가 0인 경우에 설정된다.

N은 전 48비트 결과가 비트 47 세트를 가졌을 때 설정된다(부(-))

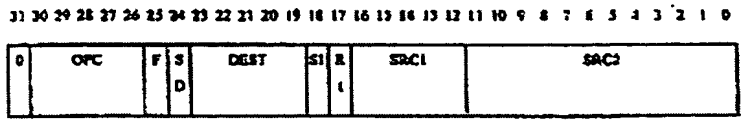
V는 (1) 수신지 레지스터가 16비트이고, 부호 결과가 16비트 레지스터 내에 맞지 않거나(범위 -2^15<=x<2^15의 범위에 없는 경우), 또는 (2) 수신지 레지스터가 32/48 비트 레지스터이고, 부호 결과가 32비트에 맞지 않는 경우에 설정된다.

C는 보존된다.

두 번째의 플래그(SZ, SN, SV, SC)는 보존된다.

논리 동작, 병렬 덧셈 및 뺄셈, 최대 및 최소, 시프트 등을 포함하는 다른 명령은 아래에 포함된다.

덧셈 및 뺄셈 명령은 2개의 레지스터를 덧셈하거나 뺄셈하고, 그 결과를 스케일한 후, 레지스터로 되돌아가서 저장한다. 오퍼런드는 부호 값으로서 취급된다. 비포화 변형에 대한 플래그 갱신은 선택적이고, 명령의 끝부분에 N을 추가함으로써 억제될 것이다.



OPC는 명령의 종류를 지정한다.

액션(OPC):

100N0	dest = (src1+src2)(->> scale)(, N)
110N0	dest = (src1-src2)(->> scale)(, N)
10001	dest = SAT((src1+src2)(->> scale))
11001	dest = SAT(src1-src2)(->> scale)
01110	dest = (src2-src1)(->> scale)
01111	dest = SAT((src2-src1)(->> scale))
101N0	dest = (src1+src2+Carry)(->> scale)(, N)
111N0	dest = (src1-src2+Carry-1)(->> scale)(, N)

연상기호:

100N0	ADD{N}	<dest>, <src1>, <src2> {,<scale>}
110N0	SUB{N}	<dest>, <src1>, <src2> {,<scale>}
10001	SADD	<dest>, <src1>, <src2> {,<scale>}
11001	SSUB	<dest>, <src1>, <src2> {,<scale>}
01110	RSB	<dest>, <src1>, <src2> {,<scale>}
01111	SRSB	<dest>, <src1>, <src2> {,<scale>}
101N0	ADC{N}	<dest>, <src1>, <src2> {,<scale>}
111N0	SBC{N}	<dest>, <src1>, <src2> {,<scale>}

어셈블러는 다음의 연산코드를 지지한다.

CMP <src1>, <src2>

CMN <src1>, <src2>

CMP는 레지스터 기록이 불가능한 플래그를 설정하는 감산이다. CMN은 레지스터 기록이 불가능한 플래그를 설정하는 가산이다.

플래그:

이들에 대하여는 위에서 설명했다.

포함 이유:

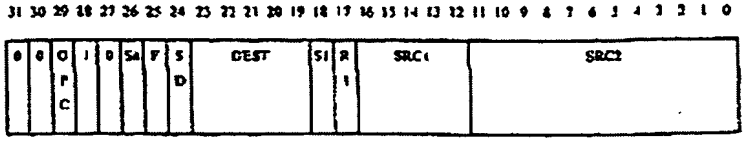
ADC는 시프트/MAX/MIN 동작을 따르는 레지스터의 하부 내에 캐리를 삽입하는데 유용하다. 또한, 그것은 32/32비트 나눗셈을 위해 사용된다. 또한, 확장된 정밀 가산에 대비한다. N 비트의 부가는 플래그의 보다

미세한 제어, 특히 케리를 나타낸다. 이것은 비트마다 2 사이클로 32/32 비트 나눗셈을 허용한다.

포화된 덧셈 및 뺄셈은 G.729 등에 필요하다.

카운터를 증가/감소시키면, RSB는 시프트를 계산하는데 유용하다(X=32-x는 공통연산이다). 포화된 RSB는 (G729에 사용된) 포화된 부정에 필요하다.

덧셈/뺄셈 누산 명령은 누산 및 스케일링/포화로 덧셈 및 뺄셈을 수행한다. 곱셈 누산 명령과는 달리, 누산기 수는 수신지 레지스터와 관계없이 지정될 수 없다. 수신지 레지스터의 하부 2개의 비트는 누산기 내부에 누적하기 위한 48비트 누산기의 개수, acc를 나타낸다. 따라서, ADDA X0,X1,X2,A0 및 ADDA A3,X1,X2,A3은 유효하지만, ADDA X1,X1,X2,A0은 유효하지 않다. 이 명령의 클래스에 의해, 결과가 레지스터로 되돌아가서 기록되어야 하고, 어떠한 수신지 필드의 기록백 인코딩도 허용되지 않다.



OPC는 명령의 종류를 지정한다. 다음에서 acc는 (DEST[1:0])이다. Sa 비트는 포화를 나타낸다.

액션(OPC):

0 dest = {SAT}(acc+(src1 + src2)){->> scale}

1 dest = {SAT}(acc+(src1 - src2)){->> scale}

연상기호:

0 {S}ADDA <dest>, <src1>, <src2>, <acc> {,<scale>}

1 {S}SUBA <dest>, <src1>, <src2>, <acc> {,<scale>}

명령 전의 S는 포화를 나타낸다.

플래그:

상기 참조

포함 이유:

ADDA(덧셈 누산) 명령은 사이클마다 누산기(예컨대, 그룹의 평균을 찾기 위해)로 정수들의 어레이의 2개의 워드를 합계하는데 유용하다. SUBA(뺄셈 누산) 명령은 (상호관계에 대한) 차의 합계를 계산하는데 유용하고, 또 그것은 2개의 나눗셈된 값을 뺄셈하고, 그 차를 제 3 레지스터와 덧셈한다.

<acc>와 다른 <dest>를 사용함으로써 라운딩과 함께 덧셈이 행해질 수 있다. 예컨대, X0=(X1+X2+16384)>>15는 A0 내에 16384를 유지함으로써 한 개의 사이클로 행해질 수 있다. 라운딩 상수와 함께 덧셈은 ADDA X0, X1, #16384, A0에 의해 행해질 수 있다.

((a_i+b_i)>>k)의 합계(완전히 공통-TrueSpeech에 사용된)의 비트의 정확한 실행에 대하여, 표준 피콜로 코드는 다음과 같다.

```
MUL   t1, a_0, b_0, ASR#k
ADD   ans, ans, t1
MUL   t2, a_1, b_1, ASR#k
ADD   ans, ans, t2
```

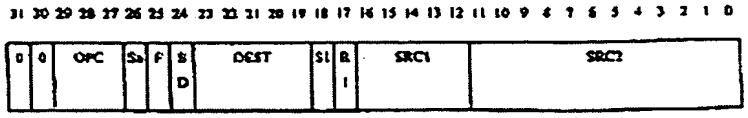
이 코드에 있어서는 2개의 문제점이 있다. 하나는 너무 길다는 것이고, 다른 하나는 48비트 정밀도에 덧셈이 없어 가드 비트가 사용될 수 없다는 것이다. 보다 좋은 해답은 ADDA를 사용하는 것이다.

```
MUL   t1, a_0, b_0, ASR#k
MUL   t2, a_1, b_1, ASR#k
```

ADDA ans, t1, t2, ans

이것은 25% 속도 증가를 주고, 48비트 정확도를 유지한다.

병렬 명령에서의 덧셈/뺄셈은 32비트 레지스터 내에서 쌍으로 보유된 2개의 부호 16비트 양에 대하여 덧셈 및 뺄셈을 행한다. 첫 번째의 조건 코드 플래그는 최상위 16비트의 결과로부터 설정되고, 두 번째의 플래그는 최하위 절반으로부터 갱신된다. 값이 하프워드 교환될 수 있더라도, 32비트 레지스터만이 이들 명령에 대한 소스로서 지정될 수 있다. 각 레지스터의 계개의 절반은 부호 값으로서 취급된다. 계산 및 스케일링은 어떤 정확도의 손실없이 행해진다. 따라서, ADDADD X0, X1, X2, ASR#1은 X0의 상부 및 하부 절반에서 정확한 평균을 생성할 것이다. 임의로 포화하는 것은 Sa 비트가 설정되어야 하는 각 명령에 대하여 준비된다.



OPC는 동작을 정의한다.

액션(OPC):

000 dest.h=(src1.h+src2.h)->>{scale},

dest.l=(src1.l+src2.l)->>{scale}

001 dest.h=(src1.h+src2.h)->>{scale},

dest.l=(src1.l-src2.l)->>{scale}

100 dest.h=(src1.h-src2.h)->>{scale},

dest.l=(src1.l+src2.l)->>{scale}

101 dest.h=(src1.h-src2.h)->>{scale},

dest.l=(src1.l-src2.l)->>{scale}

각 합계/차는 Sa 비트가 설정되면 독립하여 포화된다.

연상기호:

000	{S}ADDADD	<dest>, <src1_32>, <src2_32> {,<scale>}
001	{S}ADDSUB	<dest>, <src1_32>, <src2_32> {,<scale>}
100	{S}SUBADD	<dest>, <src1_32>, <src2_32> {,<scale>}
101	{S}SUBSUB	<dest>, <src1_32>, <src2_32> {,<scale>}

명령 전의 S는 포화를 나타낸다.

또한, 어셈블러는 되돌아가서 기록하는일 없이 표준 명령에 의해 발생된 아래의 것들을 지지한다.

CMNCMN	<dest>, <src1_32>, <src2_32> {,<scale>}
CMNCMP	<dest>, <src1_32>, <src2_32> {,<scale>}
CMPCMN	<dest>, <src1_32>, <src2_32> {,<scale>}
CMPCMP	<dest>, <src1_32>, <src2_32> {,<scale>}

플래그:

C는 2개의 상부 16 비트 절반을 더할 때 비트 15를 수행하면 설정된다.

Z는 상부 16 비트 절반의 합계가 0이면 설정된다.

N은 상부 16 비트 절반이 부(-)이면 설정된다.

V는 상부 16 비트 절반의 부호 17비트 합계가 16비트 내에 맞지 않으면 설정된다(사후 스케일).

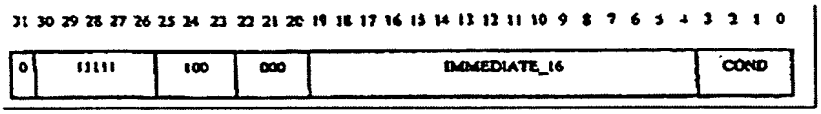
SZ, SN, SV, 및 SC는 하부 16 비트 절반에 대하여 비슷하게 설정된다.

포함 이유:

병렬 덧셈 및 뺄셈 명령은 32비트 레지스터 내에 보유된 복소수에 대한 연산을 수행하는데 유용하다. 이들 명령은 FFT 커널(kernel)에 사용된다. 이것은 또한 16비트 데이터의 벡터의 간단한 덧셈/뺄셈에 유용하여, 2개의 소자가 사이클마다 프로세스될 수 있다.

브렌치(조건적인) 명령은 제어 흐름에서의 조건적인 변화를 허용한다.

피플로는 취득한 브렌치를 실행하기 위해 3개의 사이클을 허한다.



액션:

<cond>가 첫 번째의 플래그에 따라 유지하는 경우의 오프셋에 의한 브렌치.

이 오프셋은 워드의 부호 16비트 계수이다. 현재 오프셋의 범위는 -32768~

+32767 워드로 제한된다.

수행된 어드레스 연산은

타겟 어드레스 = 브렌치 명령 어드레스 + 4 + OFFSET이다.

연상기호:

B<cond> <destination_label>

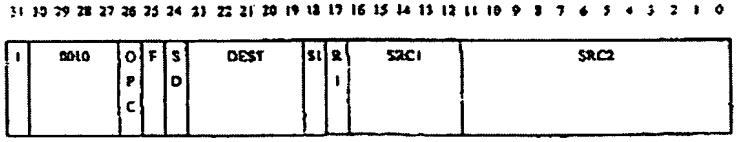
플래그:

영향을 받지 않음

포함 이유:

대부분의 루틴에서 아주 유용.

조건적인 덧셈 또는 뺄셈 명령은 src2~src1를 조건적으로 덧셈하거나 뺄셈한다.



OPC는 명령의 종류를 지정한다.

액션(OPC):

0 (캐리 설정)그 밖의 temp=src1-src2이면, temp=src1+src2
dest=temp {->> scale}

1 (캐리 설정)그 밖의 temp=src1-src2이면, temp=src1+src2
dest=temp {->> scal} 그러나 스케일이 좌측 시프트이면,
캐리의 신규값(src1-src2 또는 src1+src2로부터)은 하부로 시프트된 다.

연상기호:

0 CAS <dest>, <src1>, <src2> {,<scale>}

1 CASC <dest>, <src1>, <src2> {,<scale>}

플래그:
상기 참조

포함 이유:
조건적인 덧셈 또는 뺄셈 명령에 의해 효과적인 나눗셈 코드가 구성될 수 있다.

예 1: X1에서의 16비트 부호없는 값에 의해 X0에서의 32비트 부호없는 값을 나눈다(X0<(X1<<16) 및 X1.h=0이라는 가정 아래).

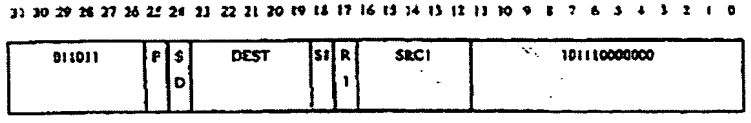
```
LSL     X1, X1, #15             ; 제수 시프트 업
SUB     X1, X1, #0              ; 캐리 플래그 설정
REPEAT #16
CASC X0, X0, X1, LSL#1
NEXT
```

루프 X0.l의 끝부분에서는 나눗셈의 몫을 보유한다. 나머지는 캐리의 값에 의존하여 X0.h로부터 회복될 수 있다.

예 2: 초기에 종료한 채로, X1에서의 32비트 정(+)의 값을 X0에서의 32비트 정(+)의 값으로 나눈다.

```
MOV             X2, #0          ; 몫을 클리어
LOG             Z0, X0          ; 비트 X0의 수가 시프트될 수 있다.
LOG             Z1, X1          ; 비트 X1의 수가 시프트될 수 있다.
SUBS            Z0, Z1, Z0              ; X1은 시프트 업하므로 1들이 매칭된다.
BLT             div_end         ; X1>X0이므로 응답은 0이다.
LSL             X1, X1, Z0              ; 주요한 것들을 매칭한다.
ADD             Z0, Z0, #1              ; 해야 할 테스트의 수.
SUBS            Z0, Z0, #0              ; 캐리를 설정한다.
REPEAT Z0
CAS             X0, X0, X1, LSL#1
ADCN            X2, X2, X2
NEXT
div_end
```

끝부분에서, X2는 몫을 보유하고, 나머지는 X0로부터 복원될 수 있다.
카운트 선행 비트 명령에 의해 데이터가 표준화될 수 있다.



액션:
dest는 공간의 개수로 설정되고, src1에서의 값은 비트 30과 다르게 비트 31에 대하여 정돈되어 좌측으로 시프트되어야 한다. 이것은 src1이 -1이거나 0이고, 31이 반환되는 특수한 경우를 제외하고 0-30 범위 내의 값이다.
연상기호:

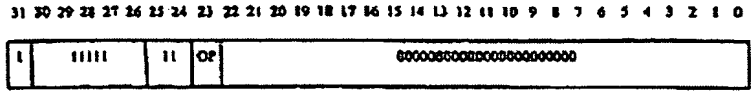
```
CLB     <dest>, <src1>
```

플래그:
Z는 결과가 0인 경우에 설정된다.
N은 클리어된다.
C는 src1이 -1 또는 0 중 어느 하나인 경우에 설정된다.
V는 보존된다.

포함 이유:

표준화에 필요한 단계.

정지 및 브레이크 포인트 명령은 피큐로 실행을 정지시키기 위해 제공된다.



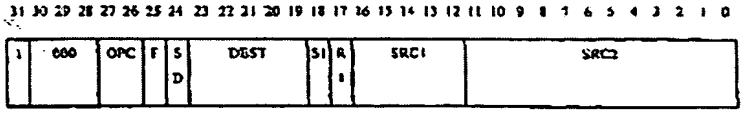
OPC는 명령의 종류를 지정한다.

액션(OPC):
0 피큐로 실행은 정지되고, 정지 비트는 피큐로 상태 레지스터에서 설정된다.
1 피큐로 실행은 정지되고, 브레이크 비트는 피큐로 상태 레지스터에서 설정되며, ARP은 브레이크 포인트가 도달되었다는 것을 알리기 위해 인터럽트된다.

연상기호:
0 HALT
1 BREAK

플래그:
영향을 받지 않음

논리 동작 명령은 32비트 또는 16비트 레지스터에 대하여 논리동작을 수행한다. 오퍼랜드는 부호없는 값으로서 처리된다.



OPC는 수행할 논리동작을 인코드한다.

액션(OPC):
00 dest=(src1 & src2) {->> scale}
01 dest=(src1 : src2) {->> scale}
10 dest=(src1 & ~src2) {->> scale}
11 dest=(src1 ^ src2) {->> scale}

연상기호:

00 AND <dest>, <src1>, <src2> {,<scale>}

01 ORR <dest>, <src1>, <src2> {,<scale>}

10 BIC <dest>, <src1>, <src2> {,<scale>}

11 EOR <dest>, <src1>, <src2> {,<scale>}

어셈블러는 다음의 연산코드를 지지한다.

TST <src1>, <src2>

TEQ <src1>, <src2>

TST는 레지스터 기록 불가능한 AND이다. TEQ는 레지스터 기록 불가능한 EOR이다.

플래그:

Z는 결과가 모두 0인 경우에 설정된다.

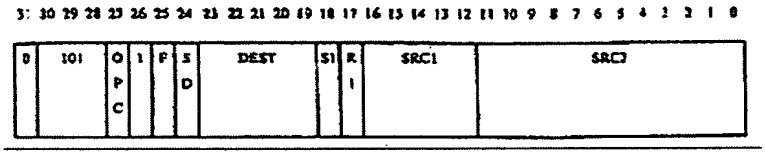
N, C, V는 보존된다.

SZ, SN, SC, SV는 보존된다.

포함 이유:

스피치 압축 알고리즘은 정보를 인코딩하기 위한 꽉 찬 비트 필드를 사용한다. 비트 마스킹 명령은 이들 필드를 추출/패킹하는 것을 돕는다.

Max 및 Min 동작명령은 최대 및 최소동작을 수행한다.



OPC는 명령의 종류를 지정한다.

액션(OPC):

0 dest=(src1 <= src2)? src1 : src2

1 dest=(src1 > src2)? src1 : src2

연상기호:

0 MIN <dest>, <src1>, <src2>

1 MAX <dest>, <src1>, <src2>

플래그:

Z는 결과가 00이면 설정된다.

N은 결과가 부(-)이면 설정된다.

C는 Max에 대하여, src2>=src1이면 설정되고(dest=src1 경우)

Min에 대하여, src2>=src1이면 설정된다(dest=src2 경우).

V는 보존된다.

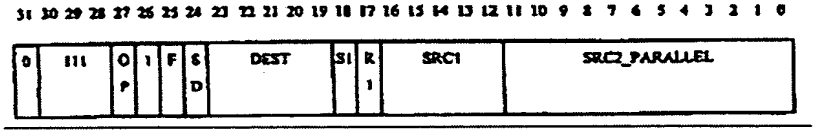
포함이유:

신호의 강도를 찾기 위해, 많은 알고리즘은 하나의 샘플을 스캔하여 샘플의 절대값의 최소/최대를 찾는다. MAX 및 MIN 등작은 이것에는 매우 중요하다. 신호에서 첫 번째 또는 마지막 최대를 찾기를 원하는지의 여부에 따라, 오퍼런드 src1 및 src2가 교환될 수 있다.

MAX X0, X0, #0은 아래로 클리핑하여 X0을 정(+)의 수로 전환할 것이다.

MIN X0, X0, #255는 상기 X0을 클리핑할 것이다. 이것은 그레픽 처리에 유용하다.

병렬명령에서의 Max 및 Min 등작은 병렬 16비트 데이터에 대하여 최대 및 최소 등작을 수행한다.



OPC는 명령의 종류를 지정한다.

액션(OPC):

0 dest.l=(src1.l<=src2.l)? src1.l : src2.l
dest.h=(src1.h<=src2.h)? src1.h : src2.h

1 dest.l=(src1.l>src2.l)? src1.l : src2.l
dest.h=(src1.h>src2.h)? src1.h : src2.h

연상기호:

0 MINMIN <dest>, <src1>, <src2>

1 MAXMAX <dest>, <src1>, <src2>

플래그:

Z는 결과의 상부 16비트가 0이면 설정된다.

N은 결과의 상부 16비트가 부(-)이면 설정된다.

C는 Max에 대하여, src2.h>=src1.h이면 설정된다(dest=src1 경우)

Min에 대하여, src2.h>=src1.h이면 설정된다(dest=src2 경우)

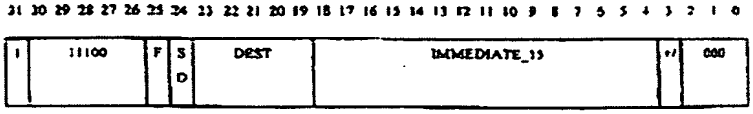
V는 보존된다.

SZ, SN, SC, SV는 하부 16비트 절반에 대하여 비슷하게 설정된다.

포함 이유:

32비트 Max 및 Min에 대하여.

이동이 긴 즉시 등작 명령에 의해 레지스터가 어떤 부호 16비트, 부호 확장된 값으로 설정될 수 있다. 이들 명령 중 2개는 32비트 레지스터를 (차례로 상부 및 하부 절반을 액세스함으로써) 어떤 값으로 설정할 수 있다. 레지스터 사이를 이동하는 것에 대하여는 선택 등작을 참조한다.



연상기호:

MOV <dest>, #<imm_16>

어셈블러는 이 MOV 명령을 이용하여 비인터로킹(non-interlocking) NOP 등작을 제공하고, 즉, NOP는 MOV,

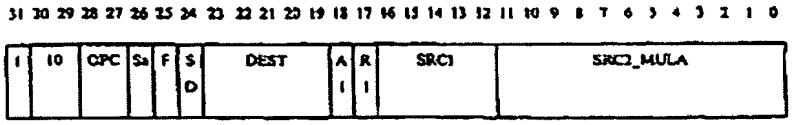
#0과 같다.

플래그:
플래그들은 영향을 받지 않는다.

포함이유:

레지스터/카운터를 초기화.

곱셈 누산 동작 명령은 누적 또는 누적처분, 스케일링 및 포화로 부호있는 곱셈을 수행한다.



필드 OPC는 명령의 종류를 지정한다.

액션(OPC):
00 dest=(acc+(src1*src2)) {->> scale}
01 dest=(acc-(src1*src2)) {->> scale}
각 경우에 있어서, Sa비트가 설정되면, 결과는 수신지에 기록되기 전에 포화된다.

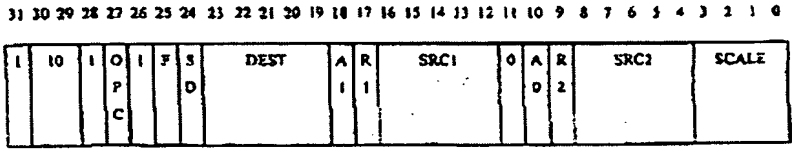
연상기호:
00 {S}MULA <dest>, <src1_16>, <src2_16>, <acc> {,<scale>}
01 {S}MULS <dest>, <src1_16>, <src2_16>, <acc> {,<scale>}
명령 전의 S는 포화를 나타낸다.

플래그:
상기 부분을 참조

포함 이유:

1개의 사이클 지속 MULA은 FIR 코드에 필요하다. MULS는 FFT 버터플라이(butterfly)에 사용된다. MULA는 또한 순환하여 곱셈하는데 유용하다. 예컨대, A0=(X0+X1+16384)>>15는 또 다른 누산기(예컨대 A1) 내에 16284를 보유함으로써 한 번의 사이클로 행해질 수 있다. 서로 다른 <dest> 및 <acc>는 또한 FFT 커널에 필요하다.

곱셈 이중 동작 명령은 부호있는 곱셈을 수행하여, 누적 또는 누적처분, 스케일링 및 포화 전에 결과를 두 배로 한다.



OPC는 명령의 종류를 지정한다.

액션(OPC):

0 dest=SAT((acc+SAT(2*src1*src2)) {->> scale})

1 dest=SAT((acc-SAT(2*src1*src2)) {->> scale})

연상기호:

0 SMLDA <dest>, <src1_16>, <src2_16>, <acc> {,<scale>}

1 SMLDS <dest>, <src1_16>, <src2_16>, <acc> {,<scale>}

플래그:

상기 부분을 참조

포함이유:

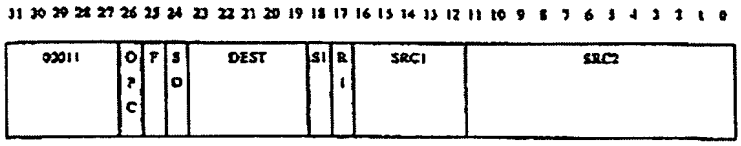
MLD 명령은 G.729 및 소수부분 연산을 이용하는 다른 알고리즘에 필요하다. 대부분이 DSP는 누산 또는 기록백 전에, 곱셈기의 출력에서 1비트의 좌측 시프트를 허용하는 소수부분 모드를 제공한다. 특정한 명령이 보다 더 프로그래밍 유연성을 제공할 때 이것을 지지한다. 일부 G 시리즈 기본 동작에 대한 명칭 등가는 다음과 같다.

L_mus => SMLDS

L_mac => SMLDA

이들은 1비트씩 좌측으로 시프트할 때 곱셈기의 포화를 이용한다. 정밀도의 손실없이, 소수부분의 곱셈-누산의 시퀀스가 요구되면, 합계가 1.15형식으로 유지된 채로 MULA가 사용될 수 있다. 필요하다면, 1.15 형식으로 전환하기 위해 좌측으로의 시프트 및 포화가 끝부분에서 사용될 수 있다.

곱셈 동작 명령은 부호있는 곱셈 및, 임의의 스케일링/포화를 수행한다. 소스 레지스터(16비트만)는 부호있는 수로서 간주된다.



OPC는 명령의 종류를 지정한다.

액션(OPC):

0 dest=(src1*src2){->>scale}

1 dest=SAT((src1*src2){->>scale}

연상기호:

0 MUL <dest>, <src1_16>, <src2>, {,<scale>}

1 SMUL <dest>, <src1_16>, <src2>{,<scale>}

플래그:

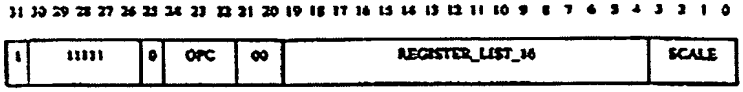
상기 부분을 참조

포함이유:

많은 프로세스는 부호가 있고, 포화되어 있는 곱셈을 요구한다.

레지스터 리스트 동작은 1세트의 레지스터에 대한 액션을 행하기 위해 사용된다. 공백 및 제로(Empty 및

Zero)명령은 루틴 전에 또는 루틴 중간에 레지스터의 선택을 리세트하기 위해 제공된다. 출력 명령을 제공하여 레지스터의 리스트의 내용을 출력 FIFO에 저장한다.



OPC는 명령의 종류를 지정한다.

액션(OPC):

000 (k=0; k<16; k++)에 대하여
레지스터 리스트의 비트 k가 설정되면, 레지스터 k는 공백이라고 표시된다.

001 (k=0; k<16; k++)에 대하여
레지스터 리스트의 비트 k가 설정되면, 레지스터 k는 0을 포함하도록 설정된다.

010 불확정

011 불확정

100 (k=0; k<16; k++)에 대하여
레지스터 리스트의 비트 k가 설정되면, (resister k->>scal)은 출력 FIFO에 기록된다.

101 (k=0; k<16; k++)에 대하여
레지스터 리스트의 비트 k가 설정되면, (resister k->>scal)은 출력 FIFO에 기록되고, 레지스터 k는 공백이라고 표시된다.

110 (k=0; k<16; k++)에 대하여
레지스터 리스트의 비트 k가 설정되면, SAT(resister k->>scal)는 출력 FIFO에 기록된다.

111 (k=0; k<16; k++)에 대하여
레지스터 리스트의 비트 k가 설정되면, SAT(resister k->>scal)는 출력 FIFO에 기록되고, 레지스터 k는 공백으로 표시된다.

연상기호:

000 EMPTY <register_list>
001 ZERO <register_list>
010 미사용
011 미사용
100 OUTPUT <register_list> {,<scale>}
101 OUTPUT <register_list>^ {,<scale>}
110 SOUTPUT <register_list> {,<scale>}
111 SOUTPUT <register_list>^ {,<scale>}

플러그:

영향을 받지 않음

예:

EMPTY {A0, A1, X0-X3}
ZERO {Y0-Y3}

OUTPUT {X0-Y1}^

어셈블러는 또한 이 신택스를 지지한다.

OUTPUT Rn

이 경우, 그것은 MOV^, Rn 명령을 이용하여 1개의 레지스터를 출력할 것이다.

EMPTY 명령은 공백의 모든 레지스터가 유효한 데이터를 포함할 때까지(즉, 공백이 아닐 때까지) 정지할 것이다.

레지스터 리스트 동작은 리맵핑 REPEAT 루프 내에 사용되지 않아도 된다.

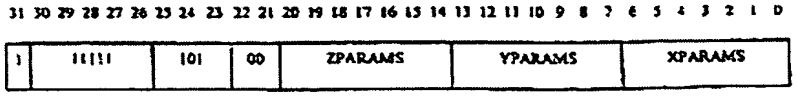
OUPUT 명령은 단지 출력을 위해 8개의 레지스터까지만 지정할 수 있다.

포함이유:

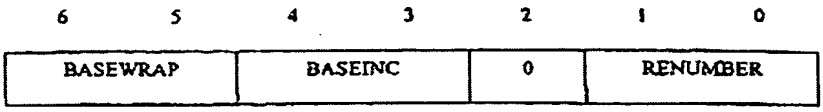
루틴이 종료된 후에, 다음 루틴은 모든 레지스터가 공백이라고 생각하여, AR버으로부터 데이터를 수신한다. EMPTY 명령은 이것을 수행하는데 필요하다. FIR 또는 다른 필터를 수행하기 전에, 모든 누산기 및 부분적인 결과를 0으로 되어야 한다. ZERO 명령은 이것을 돕는다. 양쪽은 일련의 단일 레지스터 이동을 교환함으로써 코드 밀도를 향상시키도록 설계되어 있다. OUTPUT 명령은 일련의 MOV^, Rn 명령을 교환함으로써 코드밀도를 향상시키도록 포함된다.

리맵핑 파라미터 이용 명령 RMOV은 사용자가 규정된 레지스터 리맵핑 파라미터의 구성을 허용하도록 제공된다.

명령 디코딩은 다음과 같다.



각 PARAMS 필드는 다음의 엔트리로 구성된다.



이들 엔트리의 의미는 아래에 설명되어 있다.

파라미터	설명
RENUMBER	리맵핑을 행하기 위한 16비트 레지스터의 수는 값 0, 2, 4, 8을 갖는다. RENUMBER아래의 레지스터는 리맵핑되고, 상기의 이들은 직접 액세스된다.
BASEINC	베이스 포인터의 양은 각 루프의 끝부분에서 증가된다. 값 1, 2, 또는 4를 갖는다.
BASEWRAP	베이스 중첩 계수는 값 2, 4, 8을 갖는다.

연상기호:

RMOV <PARAMS>, [<PARAMS>]

<PARAMS> 필드는 다음의 형식을 갖는다.

<PARAMS> ::= <BANK><BASEINC>n<RENUMBER>

w<BASEWRAP>

<BANK> ::=[X:Y:Z]

<BASEINC> ::=[++:+1:+2:+4]

<RENUMBER> ::=[0:2:4:8]

<BASEWRAP> ::=[2:4:8]

리맵핑이 활동중인 동안에 RMOV 명령이 사용되면, 작용은 UNPREDICTABLE이다.

플러그:

영향을 받지 않음

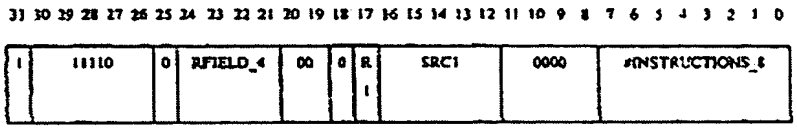
반복명령은 하드웨어에서 4개의 제로 사이클 루프를 제공한다. REPEAT 명령은 신규 하드웨어 루프를 제한한다. 피클로는 제 1 REPEAT 명령에 대하여 하드웨어 루프 0을 사용하고, 제 1 반복명령 등 내에 네스트된 REPEAT 명령에 대하여 하드웨어 루프 1을 사용한다. REPEAT 루프는 정확히 네스트되어야 된다. 4보다 더 큰 깊이에 루프를 네스트하려고 시도하면, 등작은 예측될 수 없다.

각 REPEAT 명령은 (REPEAT 명령을 즉시 따르는) 루프에서 명령의 수와 횟수를 지정하여 루프를 순환한다(어느 쪽이든 하나의 상수이거나, 피클로 레지스터로부터 판독된다).

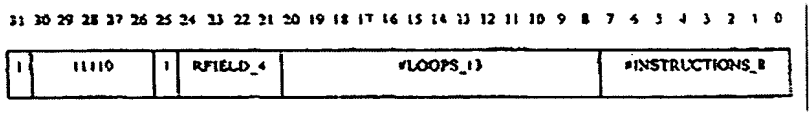
루프에서의 명령의 수가 작으면(1 또는 2), 피클로는 여분의 사이클을 취하여 루프 업을 설정한다.

루프 카운트가 레지스터 지정되는 경우, 하부 16비트만이 중요하고, 그 수는 부호가 없는 것이라고 갖주되더라도, 32비트 액세스가 포함된다(S1=1). 루프 카운트가 0이면, 루프의 액션은 제한되지 않는다. 루프 카운트의 복사를 이용하여 루프에 영향을 미치지 않고 레지스터를 즉시 다시 이용할 수 있다(또는 재충전할 수 있다).

REPEAT 명령은 메카니즘을 제공하여 레지스터 오퍼랜드가 루프 내에 지정되는 방법을 변경한다. 레지스터 지정된 루프의 수로 상기 REPEAT의 인코딩에 대하여 상세히 설명한다.



고정된 루프의 수로 REPEAT 인코딩:



RFIELD 오퍼랜드는 루프 내부에 이용하기 위해 16개의 리맵핑 파라미터 구성 중 어느 것인가를 지정한다.

RFIELD	리맵핑 등작
0	리맵핑이 수행되지 않음
1	사용자 정의 리맵핑
2..15	리맵핑 구성 TBD 프리세트

어셈블리는 하드웨어 루프를 정의하는 2개의 연산코드 REPEAT 및 NEXT를 제공한다. REPEAT는 루프의 처음에 놓이고, NEXT는 루프의 끝부분에서 범위를 정하기 때문에, 어셈블러는 루프 본체에서 명령의 수를 계산할 수 있다. REPEAT에 대하여는, 오직 상수 또는 레지스터 중 어느 하나로서 루프의 수를 지정해야 한다. 예를 들면,

```
REPEAT      X0
MULA            A0, Y0.l, Z0.l, A0
MULA            A0, Y0.h^, Z0.h^, A0
NEXT
```

이것은 2개의 MULA 명령을 X0회 실행할 것이다. 또한, 이것은

```
REPEAT      #10
MULA                A0, X0^, Y0^, A0
NEXT
```

10회의 곱셈 누산을 수행할 것이다.

어셈블러는 다음의 신택스를 지정한다.

REPEAT #Iterations [, <PARAMS>]

REPEAT에 대하여 사용하기 위한 리맵핑 파라미터를 지정한다. 요구된 리맵핑 파라미터가 소정 세트의 파라미터 중 하나와 같으면, 적당한 REPEAT 인코딩이 사용된다. 그것이 없으면, 어셈블러는 사용자 정의 파라미터를 로드하기 위해 REPEAT 명령 전에 오는 RMOV를 발생할 것이다. RMOV 명령 및 리맵핑 파라미터 형식의 상세한 것에 대하여는 상기 부분을 참조.

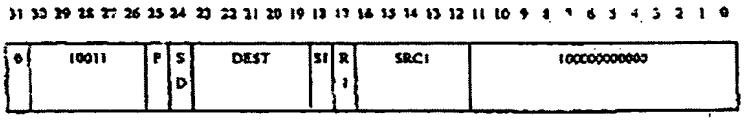
루프 반복의 수가 0이면, REPEAT의 액션은 UNPREDICTABLE이다.

명령 필드의 수가 0으로 설정되면, REPEAT의 액션은 UNPREDICTABLE이다.

브렌치인 1개의 명령만으로 구성된 루프는 UNPREDICTABLE 작용을 가질 것이다.

REPEAT 루프의 경계 내에 있고, 그 루프의 경계밖에 있는 브렌치는 UNPREDICTABLE이다.

포화 절대 명령은 포화된 소스 1의 절대 값을 계산한다.



액션:

dest=SAT((SRC1>=0)? SRC1 : -SRC1). 이 값은 항상 포화된다.

특히, 0x80000000은 0x7fffffff 및 NOT 0x80000000!이다.

연상기호:

SABS <dest>, <src1>

플래그:

Z는 결과가 0이면 설정된다.

N은 보존된다.

C는 src1<0이면 설정된다(dest=-src1 경우)

V는 포화가 발생했으면 설정된다.

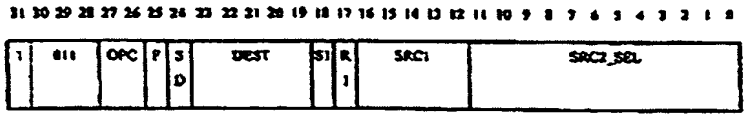
포함이유:

많은 DSP 애플리케이션에 유용.

선택 동작(조건적인 이동)은 소스 1 또는 소스 2 중 어느 하나를 수신지 레지스터 내로 조건부로 이동시키는 기능을 한다. 선택은 항상 이동과 같다. 또한 병렬 덧셈/뺄셈 후에 사용하기 위한 병렬 동작이 있다.

양쪽 소스 오퍼런드는 실행 이유의 명령에 의해 판독되어, 어느 한쪽의 소스 오퍼랜드가 공백이면, 그 명

령은 오퍼런드가 정확히 요구되는지의 여부에 관계없이 정지할 것이다.



OPC는 명령의 종류를 지정한다.

액션(OPC):

00 <cond>가 첫 번째의 플래그에 대하여 성립하면, dest=src1
아니면 dest=src2.

01 <cond>가 첫 번째의 플래그에 대하여 성립하면, dest.h=src1.h
아니면 dest.h=src2.h,
<cond>가 두 번째의 플래그에 대하여 성립하면, dest.l=src1.l
아니면 det.l=src2.l

10 <cond>가 첫 번째의 플래그에 대하여 성립하면, dest.h=src1.h
아니면 dest.h=src2.h
<cond>가 두 번째의 플래그에 대하여 실패하면, dest.l=src1.l
아니면 dest.l=src2.l

11 보류

연상기호

00	SEL<cond>	<dest>, <src1>, <src2>
01	SELTT<cond>	<dest>, <src1>, <src2>
10	SELTF<cond>	<dest>, <src1>, <src2>
11	미사용	

레지스터가 재충전을 위해 표시되면, 레지스터는 비조건부로 재충전된다. 또한 어셈블러는 다음의 연상기호를 제공한다.

MOV<cond>	<dest>, <src1>
SELFT<cond>	<dest>, <src1>, <src2>
SELFF	<cond><dest>, <src1>, <src2>

MOV<cond> A, B는 SEL<cond> A, B, A와 같다. SELFT 및 SELFF는 src1 및 src2를 교환하여 SELTF, SELTT를 사용함으로써 획득된다.

플래그:

모든 플래그가 보존되어, 선택의 시퀀스가 행해질 것이다.

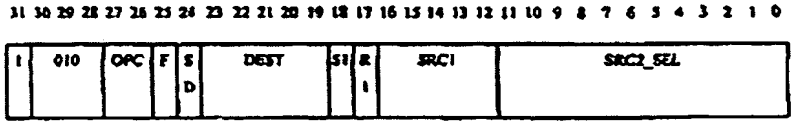
포함이유:

브렌치에 의지하지 않고 인라인을 간단히 결정하기 위해 사용된다. 비터비(Viterbi) 알고리즘에 의해 그리고 샘플 또는 가장 큰 소자에 대한 벡터를 스캐닝할 때 사용된다.

시프트 동작 명령은 좌측, 우측 논리적 시프트, 우측 연산 시프트를 제공하고, 특정한 양만큼 순환한다. 시프트 양은 레지스터 내용의 하부 8비트로부터 얻은 -128 내지 +127의 부호있는 정수 또는 +1~+31 범위내의 즉치라고 간주된다. 부(-)의 양의 시프트는 ABS(시프트 양)에 의해 반대방향으로 시프트를 일으킨다.

입력 오퍼랜드는 32비트로 확장된 부호이고, 그 결과의 32비트 출력은 기록백 전에 48비트로 확장된 부호

이므로, 48비트 레지스터로의 기록은 현저히 작용한다.



OPC는 명령의 종류를 지정한다.

액션(OPC):

00 dest=(src2>=0) ? src1 << src2 : src1 >> -src2

01 dest=(src2>=0) ? src1 >> src2 : src1 << -src2

10 dest=(src2>=0) ? src1 ->> src2 : src1 << -src2

11 dest=(src2>=0) ? src1 ROR src2 : src1 ROL -src2

연상기호:

00 ASL <dest>, <src1>, <src2_16>

01 LSR <dest>, <src1>, <src2_16>

10 ASR <dest>, <src1>, <src2_16>

11 ROR <dest>, <src1>, <src2_16>

플래그:

Z는 결과가 00이면 설정된다.

N은 결과가 부(-)이면 설정된다.

V는 보존된다.

C는 시프트 아웃된 마지막 비트의 값으로 설정된다(ARM에서와 같이).

레지스터 지정된 시프트의 작용은 다음과 같다.

- 32에 의한 LSL은 결과 0을 갖고, C는 src1의 비트 0으로 설정.
- 32 이상에 의한 LSL은 결과 0을 갖고, C는 0으로 설정.
- 32에 의한 LSR은 결과 0을 갖고, C는 src1의 비트 31로 설정.
- 32 이상에 의한 LSR은 결과 0을 갖고, C는 0으로 설정.
- 32 이상의 ASR은 src1의 비트 31로 충전된 결과를 갖고, C는 src1의 비트 31과 같다.
- 32에 의한 ROR은 src1과 같은 결과를 갖고, C는 src1의 비트 31로 설정.
- 32보다 큰 n에 의한 ROR은 동일한 결과를 나타낼 것이고, n-32에 의한 ROR로서 수행할 것이기 때문에, 그 양이 1~32 범위 내에 있을 때까지 n으로부터 32를 반복하여 뺄셈을 할 것이다.

포함 이유:

많은 2에 의한 곱셈/나눗셈. 비트 및 필드 추출. 직렬 레지스터.

규정되지 않은 명령은 명령 세트 리스팅에서 세트 아웃된다. 그들 실행은 피콜로로 하여금 실행을 정지하게 할 것이고, 상태 레지스터에서 1비트를 설정할 것이며, (제어 레지스터에서의 E비트가 클리어되었더라도) 그 자체를 디스에이블할 것이다. 이것에 의해 명령 세트의 앞으로의 어떤 확장이 트랩되어 현존하는 실행에 대하여 임의로 에뮬레이트될 수 있다.

ARM으로부터의 피콜로 상태를 엑세스하는 것은 다음과 같다. 상태 엑세스 모드를 사용하여 피콜로의 상태를 관찰/변경한다. 이 메카니즘은 아래의 2개의 목적을 위해 제공된다.

- 문맥 전환
- 디버그(debug)

피큘로는 PSTATE 명령을 실행함으로써 상태 액세스 모드에 놓인다. 이 모드에 의해 모든 피큘로 상태가 STC 및 LDC 명령의 시퀀스로 세이브 및 복원될 수 있다. 상태 액세스 모드 내에 놓이면, 피큘로 코프로세서 IO PICCOLO1의 사용을 변경하여, 피큘로의 상태를 액세스할 수 있다. 피큘로 상태에 대한 7개의 뱅크가 있다. 특정한 뱅크 내의 데이터가 모두 단일의 LDC 또는 STC로 로드 및 저장될 수 있다.

뱅크 0: 전용 레지스터

- 피큘로 IO 레지스터(관독 전용)의 값을 포함하는 1개의 32비트 워드.
- 제어 레지스터의 상태를 포함하는 1개의 32비트 워드.
- 상태 레지스터의 상태를 포함하는 1개의 32비트 워드.
- 프로그램 카운터의 상태를 포함하는 1개의 32비트 워드.

뱅크 1: 범용 레지스터(GPR).

- 범용 레지스터 상태를 포함하는 16개의 32비트 워드.

뱅크 2: 누산기

- 누산기 레지스터의 상부 32비트를 포함하는 4개의 32비트 워드다(GPR 상태와 N.B. 복사는 복원 목적을 위해 필요하고, 다른 방법으로 레지스터 뱅크에 대한 또 다른 기록 인에이블을 포함할 것이다).

뱅크 3: 레지스터/피큘로 ROB/출력 FIFO 상태.

- 레지스터가 재충전을 위해(각 32비트 레지스터에 대하여 2비트) 표시되는 것을 나타내는 1개의 32비트 워드.
- ROB 태그의 상태를 포함하는 8개의 32비트 워드(비트 7-0 내에 저장된 7개의 7비트 항목).
- 정렬되지 않는 ROB 레처의 상태를 포함하는 3개의 32비트 워드(비트 17-0)
- 출력 시프트 레지스터 내의 슬롯이 유효한 데이터를 포함하는 것을 나타내는 1개의 32비트 워드(비트 4는 공백을 나타내고, 비트 3-0은 사용된 엔트리의 수를 인코드한다).
- 출력 FIFO 유지 레처의 상태를 포함하는 1개의 32비트 워드(비트 17-0).

뱅크 4: ROB 입력 데이터.

- 8개의 32비트 데이터 값.

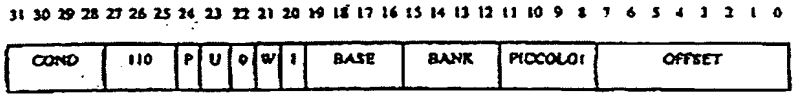
뱅크 5: 출력 FIFO 데이터.

- 8개의 32비트 데이터 값.

뱅크 6: 루프 하드웨어.

- 루프 시작 어드레스를 포함하는 4개의 32비트 워드.
- 루프 종료 어드레스를 포함하는 4개의 32비트 워드.
- 루프 카운트를 포함하는 4개의 32비트 워드(비트 15-0).
- 사용자 정의 리맵핑 파라미터 및 다른 리맵핑 상태를 포함하는 1개의 32비트 워드.

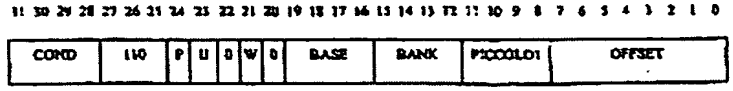
LDC 명령은 피큘로가 상태 액세스 모드에 있을 때 피큘로 상태를 로드하기 위해 사용된다. BANK 필드는 뱅크가 로드되어 있다는 것을 지정한다.



다음의 시퀀스는 레지스터 R0 내의 어드레스로부터 피큘로 상태를 모두 로드할 것이다.

```
LDP     B0, [R0], #16 ! ; 전용 레지스터
LDP     B1, [R0], #64 ! ; 범용 레지스터를 로드
LDP     B2, [R0], #16 ! ; 누산기 로드
LDP     B3, [R0], #56 ! ; 레지스터/ROB/FIFI 상태를 로드
LDP     B4, [R0], #32 ! ; ROB 데이터를 로드
LDP     B5, [R0], #32 ! ; 출력 FIFO 데이터를 로드
LDP     B6, [R0], #52 ! ; 루프 하드웨어를 로드
```

STC 명령은 피콜로가 상태 액세스 모드에 있을 때 피콜로 상태를 저장하기 위해 사용된다. BANK 필드는 뱅크가 저정되어 있다는 것을 지정한다.



다음의 시퀀스는 레지스터 R0 내의 어드레스에 피콜로 상태를 모두 저장할 것이다.

```
STP     B0, [R0], #16 ! ; 전용 레지스터를 세이브
STP     B1, [R0], #64 ! ; 범용 레지스터를 세이브
STP     B2, [R0], #16 ! ; 누산기를 세이브
STP     B3, [R0], #56 ! ; 레지스터/ROB/FIFO 상태를 세이브
STP     B4, [R0], #32 ! ; ROB 데이터를 세이브
STP     B5, [R0], #32 ! ; 출력 FIFO 데이터를 세이브
STP     B6, [R0], #52 ! ; 루프 하드웨어를 세이브
```

디버그 모드- 피콜로는 ARM에 의해 지지된 것으로서 동일한 디버그 메카니즘, 즉 Demon 및 Angel에 의한 소프트웨어와, ICE가 삽입된 하드웨어에 응답해야 한다. 다음의 피콜로 시스템을 디버그하기 위한 몇 개의 메카니즘이 있다.

- ARM 명령 브레이크 포인트(breakpoint).
- 데이터 브레이크 포인트(워치 포인트(watchpoint)).
- 피콜로 명령 브레이크 포인트.
- 피콜로 소프트웨어 브레이크 포인트.

ARM 명령 및 데이터 브레이크 포인트는 ARM 삽입된 ICE 모듈에 의해 처리되고; 피콜로 명령 브레이크 포인트는 피콜로 삽입된 ICE 모듈에 의해 처리되며; 피콜로 소프트웨어 브레이크 포인트는 피콜로 코어에 의해 처리된다.

하드웨어 브레이크 포인트 시스템은 ARM 및 피콜로가 브레이크 포인트되도록 구성할 수 있을 것이다.

소프트웨어 브레이크 포인트는 피콜로로 하여금 실행을 정지하게 하는 피콜로 명령(정지 또는 브레이크)에 의해 처리되고, 디버그 모드(상태 레지스터 세트에서의 8 비트)를 입력하고, (피콜로가 PDISABLE 명령으로 디스에이블되었던 것처럼) 그 자체를 디스에이블한다. 프로그램 카운터는 유효한 상태로 있어, 브레이크 포인트의 어드레스가 복원될 수 있다. 피콜로는 더 이상 명령을 실행하지 않을 것이다.

단일의 스테핑 피콜로는 피콜로 명령 스트림에 대하여 브레이크 포인트한 후 브레이크 포인트를 설정함으로써 행해질 것이다.

소프트웨어 디버그- 피콜로에 의해 제공된 기본 기능은 상태 액세스 모드에 있을 때 코프로세서 명령을 통해서 메모리 모든 상태를 로드 및 세이브하는 능력이다. 이것에 의해 디버거가 모든 상태를 메모리에 세이블하고, 그것을 판독 및/또는 경신하며, 그것을 피콜로에 재저장할 수 있다. 피콜로 저장 상태 메카니즘은 비파괴적일 것이고, 즉 말하자면 피콜로의 상태를 저장하는 작용이 피콜로의 어떤 내부 상태를 변조하지 않을 것이다. 이것은 피콜로가 다시 그것을 재저장하지 않고 그것의 상태를 덤핑한 후 재개될 수 있다는 것을 의미한다.

피콜로 캐시의 상태를 찾는 메카니즘이 결정될 것이다.

하드웨어 디버그-하드웨어 디버그는 피콜로의 코프로세서 인터페이스에 대한 스캔 체인에 의해 용이하게 될 것이다. 그 때에 피콜로는 상태 액세스 모드에 놓일 것이고, 스캔 체인을 통해서 조사된/변경된 상태를 가질 것이다.

피콜로 상태 레지스터는 그것이 브레이크 포인트 명령을 실행했다는 것을 나타내기 위해 단일 비트를 포함한다. 브레이크 포인트된 명령이 실행될 때, 피콜로는 상태 레지스터에서 B비트를 설정하고, 실행을 정지한다. 피콜로에 문의하기 위해, 디버거는 피콜로를 인에이블해야 하고, 다음의 액세스가 발생할 수 있기 전에 그것의 제어 레지스터에 기록함으로써 그것을 상태 액세스 모드에 둔다.

도 4는 선택된 레지스터의 적당한 절반을 피콜로 데이터경로로 전환하기 위해 Hi/Lo 비트 및 사이즈 비트에 응답하는 멀티플렉서 배치를 나타낸다. 사이즈 비트가 16비트를 나타내면, 부호 확장회로는 적당하게 0 또는 1로 데이터경로의 최고위 비트를 매립한다.

(57) 청구의 범위

청구항 1. 처리되어야 할 데이터 항목들을 저장하는 복수개의 레지스터와,

상기 복수개의 레지스터 내에 저장된 데이터 항목들에 인가되어야 할 명령을 처리하는 프로세서와,

사전에 선택된 명령 세트 내의 논리 레지스터 레퍼런스를, 상기 프로세서에 의해 처리되어야 할 데이터 항목을 포함하는 레지스터를 식별하는 물리 레지스터 레퍼런스로 변환하는 레지스터 리맵핑 논리를 구비한 것을 특징으로 하는 데이터 처리장치.

청구항 2. 제 1 항에 있어서,

레지스터 리맵핑 논리는 상기 사전에 선택된 명령 세트 전에 처리되도록 배치된 리맵핑 명령에 의해 구성될 수 있는 것을 특징으로 하는 데이터 처리장치.

청구항 3. 제 1 항 또는 제 2 항에 있어서,

상기 사전에 선택된 명령 세트로 이루어진, 반복되어야 할 명령의 범위를 정하는 반복명령과,

반복명령을 관리하고, 레지스터 리맵핑 논리를 주기적으로 갱신하도록 배치된 루프 하드웨어를 더 구비한 것을 특징으로 하는 데이터 처리장치.

청구항 4. 제 3 항에 있어서,

루프 하드웨어는 명령의 범위가 반복될 때마다 레지스터 리맵핑 논리를 갱신하도록 배치된 것을 특징으로 하는 데이터 처리장치.

청구항 5. 제 3 항 또는 제 4 항에 있어서,

반복명령은 레지스터 리맵핑 논리를 구성하기 위해 사용된 1개 또는 그 이상의 리맵핑 파라미터를 포함한 것을 특징으로 하는 데이터 처리장치.

청구항 6. 제 5 항에 있어서,

리맵핑 명령은 상기 1개 또는 그 이상의 리맵핑 파라미터를 이용하여 레지스터 리맵핑 논리를 구성하기 위해 반복명령을 실행하기 전에 행해지도록 배치되고, 반복명령에 의해 정해진 상기 명령의 범위 내에 포함되지 않은 것을 특징으로 하는 데이터 처리장치.

청구항 7. 제 5 항 또는 제 6 항에 있어서,

각각이 레지스터 리맵핑 논리에 대한 리맵핑 구성을 한정하는 적어도 한 개의 소정의 리맵핑 파라미터의 세트를 저장하는 저장수단을 더 구비하고, 상기 반복명령 내에 포함된 상기 1개 또는 그 이상의 리맵핑 파라미터가 상기 소정의 리맵핑 파라미터의 세트 중 하나에 대응하면, 레지스터 리맵핑 논리에 대한 대응하는 리맵핑 구성이 실행되어야 할 리맵핑 명령을 요구하지 않고 사용되는 것을 특징으로 하는 데이터 처리장치.

청구항 8. 제 5 항 내지 제 7 항 중 어느 한 항에 있어서,

제 1 리맵핑 파라미터는 레지스터 리맵핑 논리에 의해 리맵핑되기 쉬운 상기 복수개의 레지스터의 다수를 식별하는 것을 특징으로 하는 데이터 처리장치.

청구항 9. 제 5 항 내지 제 8 항 중 어느 한 항에 있어서,

레지스터 리맵핑 논리는 상기 복수개의 레지스터의 다수에 대하여 리맵핑을 행하도록 배치되고, 베이스 포인터는 논리 레지스터 레퍼런스에 부가되어야 할 오프셋 값으로서 레지스터 리맵핑 논리에 의해 사용되며, 반복명령은 소정의 간격으로 베이스 포인터를 증가시키는 값을 식별하는 제 2 리맵핑 파라미터를 포함한 것을 특징으로 하는 데이터 처리장치.

청구항 10. 제 9 항에 있어서,

반복명령은 제 1 증첩 값을 제공하는 제 3 리맵핑 파라미터를 포함하고, 베이스 포인터를 증가시키는 동안, 베이스 포인터가 제 1 증첩 값과 같게 되거나 초과하게 되면, 베이스 포인터의 증분이 신규 오프셋 값과 겹치는 것을 특징으로 하는 데이터 처리장치.

청구항 11. 제 9 항 또는 제 10 항에 있어서,

반복명령은 제 2 증첩 값을 제공하는 제 4 리맵핑 파라미터를 포함하고, 베이스 포인터를 부가하여 형성된 레지스터 레퍼런스 및 논리 레지스터 레퍼런스가 제 2 증첩 값과 같거나 초과하면, 이 레지스터 레퍼런스는 신규 레지스터 레퍼런스와 겹치는 것을 특징으로 하는 데이터 처리장치.

청구항 12. 선행하는 청구항 중 어느 한 항에 있어서,

상기 복수개의 레지스터는 레지스터의 뱅크를 포함하고, 레지스터 리맵핑 논리는 특정한 뱅크 내의 다수의 레지스터에 대하여 리맵핑을 행하도록 배치된 것을 특징으로 하는 데이터 처리장치.

청구항 13. 제 12 항에 있어서,

레지스터 리맵핑 논리는 레지스터의 각 뱅크에 대하여 독립적으로 리맵핑을 행하도록 배치된 것을 특징으로 하는 데이터 처리장치.

청구항 14. 선행하는 청구항 중 어느 한 항에 있어서,

명령은 각각이 논리 레지스터 레퍼런스로 이루어진 복수의 오퍼랜드를 포함하고, 레지스터 리맵핑 논리는 각 오퍼랜드에 대하여 독립적으로 리맵핑을 행하도록 배치된 것을 특징으로 하는 데이터 처리장치.

청구항 15. 선행하는 청구항 중 어느 한 항에 있어서,

상기 장치는 디지탈 신호처리장치인 것을 특징으로 하는 데이터 처리장치.

청구항 16. 데이터 처리장치를 동작시키는 방법에 있어서,

(a) 처리되어야 할 데이터 항목을 복수개의 레지스터 내에 저장하는 단계와,

(b) 명령을 처리하기 위해 요구된 1개 또는 그 이상의 데이터 항목을 복수개의 레지스터로부터 검색하는 단계와,

(c) 검색된 상기 1개 또는 그 이상의 데이터 항목을 이용하여 명령을 처리하는 단계를 구비하고,

상기 검색단계(b)는 사전에 선택된 명령 세트에 대하여, 상기 사전에 선택된 명령 세트 내의 논리 레지스터 레퍼런스를, 상기 처리단계(c)에서 요구된 데이터 항목을 포함한 레지스터를 식별하는 물리 레지스터 레퍼런스로 변환하는 부가적인 단계를 구비한 것을 특징으로 하는 방법.

청구항 17. 제 16 항에 있어서,

사전에 선택된 명령 세트로 이루어진, 반복되어야 할 명령의 범위를 정하는 단계와, 루프 하드웨어를 사용하여, 반복명령을 관리하고, 논리 레지스터 레퍼런스를 물리 레지스터 레퍼런스로 변환하는 상기 변환단계에서 사용된 레지스터 리맵핑 논리를 주기적으로 갱신하는 단계를 더 구비한 것을 특징으로 하는 방법.

청구항 18. 제 17 항에 있어서,

루프 하드웨어는 명령의 범위가 반복될 때마다 레지스터 리맵핑 논리를 갱신하도록 배치된 것을 특징으로 하는 방법.

청구항 19. 제 17 항 또는 제 18 항에 있어서,

반복되어야 할 명령의 범위를 정하는 단계는 레지스터 리맵핑 논리를 구성하기 위해 사용되어야 할 1개 또는 그 이상의 리맵핑 파라미터를 한정하는 단계를 포함한 것을 특징으로 하는 방법.

청구항 20. 제 19 항에 있어서,

리맵핑 명령은 상기 1개 또는 그 이상의 리맵핑 파라미터를 이용하여 레지스터 리맵핑 논리를 구성하기 위해, 반복되어야 할 명령의 범위를 실행하기 전에 행해지도록 배치된 것을 특징으로 하는 방법.

청구항 21. 제 19 항 또는 제 20 항에 있어서,

각각이 상기 레지스터 리맵핑 논리에 대한 리맵핑 구성을 한정하는 적어도 한 개의 소정의 리맵핑 파라미터의 세트를 저장하는 단계와,

상기 1개 또는 그 이상의 리맵핑 파라미터가 소정의 리맵핑 파라미터의 상기 세트 중 하나에 대응하면, 실행되어야 할 리맵핑 명령을 요구하지 않고 레지스터 리맵핑 논리에 대한 대응하는 리맵핑 구성을 이용하는 단계를 더 구비한 것을 특징으로 하는 방법.

청구항 22. 제 16 항 내지 제 21 항 중 어느 한 항에 있어서,

상기 복수개의 레지스터는 레지스터의 뱅크를 포함하고, 상기 방법은 특정한 뱅크 내의 다수의 레지스터에 대하여 리맵핑을 행하는 단계를 구비한 것을 특징으로 하는 방법.

청구항 23. 제 22 항에 있어서,

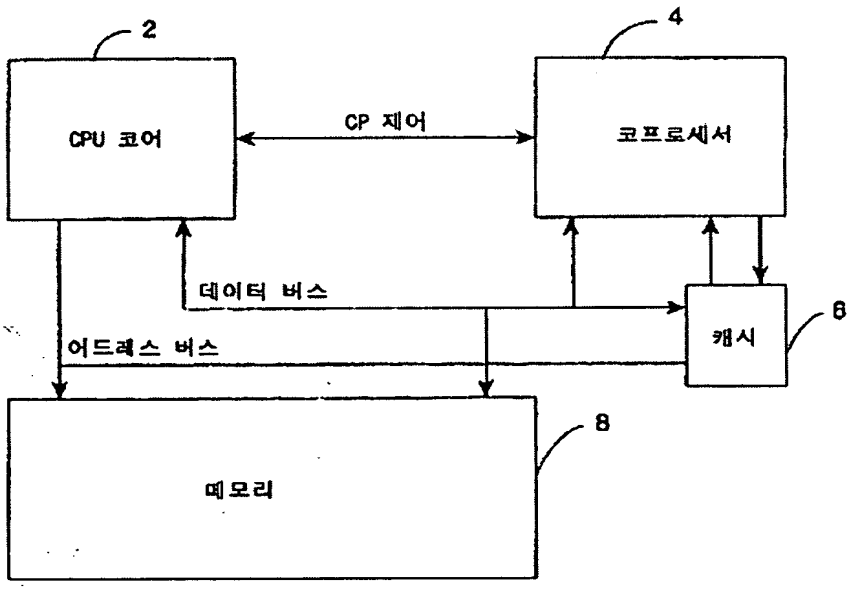
레지스터의 각 뱅크에 대하여 독립적으로 리맵핑을 행하는 단계를 더 구비한 것을 특징으로 하는 방법.

청구항 24. 제 16 항 내지 제 23 항 중 어느 한 항에 있어서,

명령은 각각이 논리 레지스터 레퍼런스로 이루어진 복수의 오퍼랜드를 포함하고, 상기 방법은 각 오퍼랜드에 대하여 독립적으로 리맵핑을 행하는 단계를 더 구비한 것을 특징으로 하는 방법.

도면

도면1

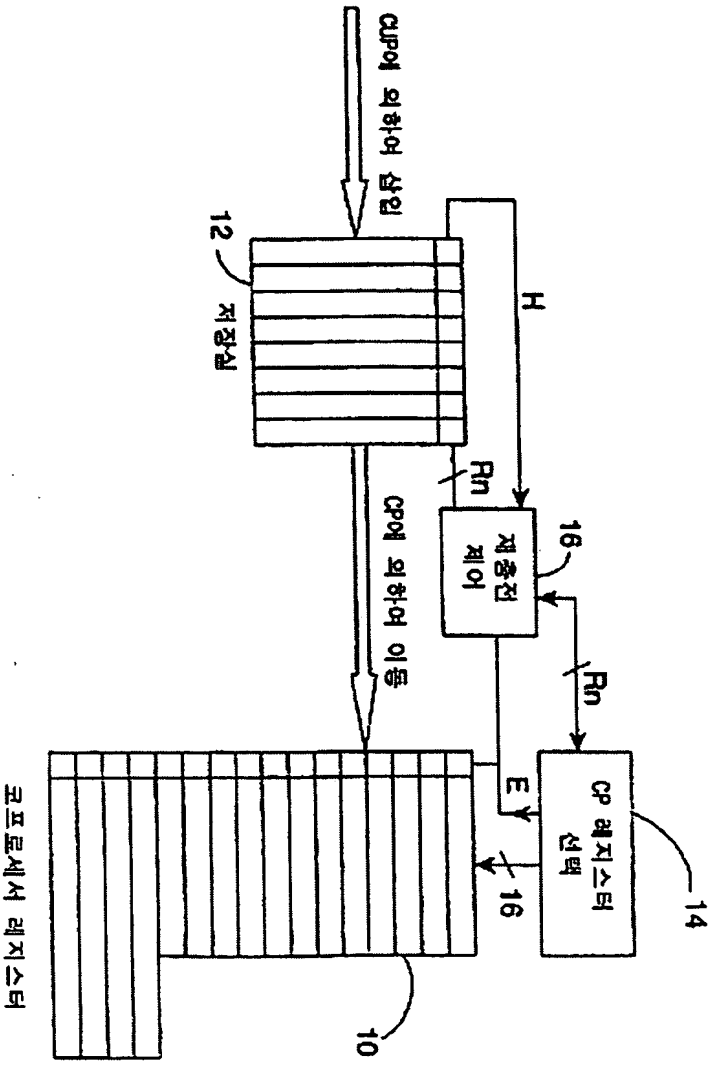


도면7

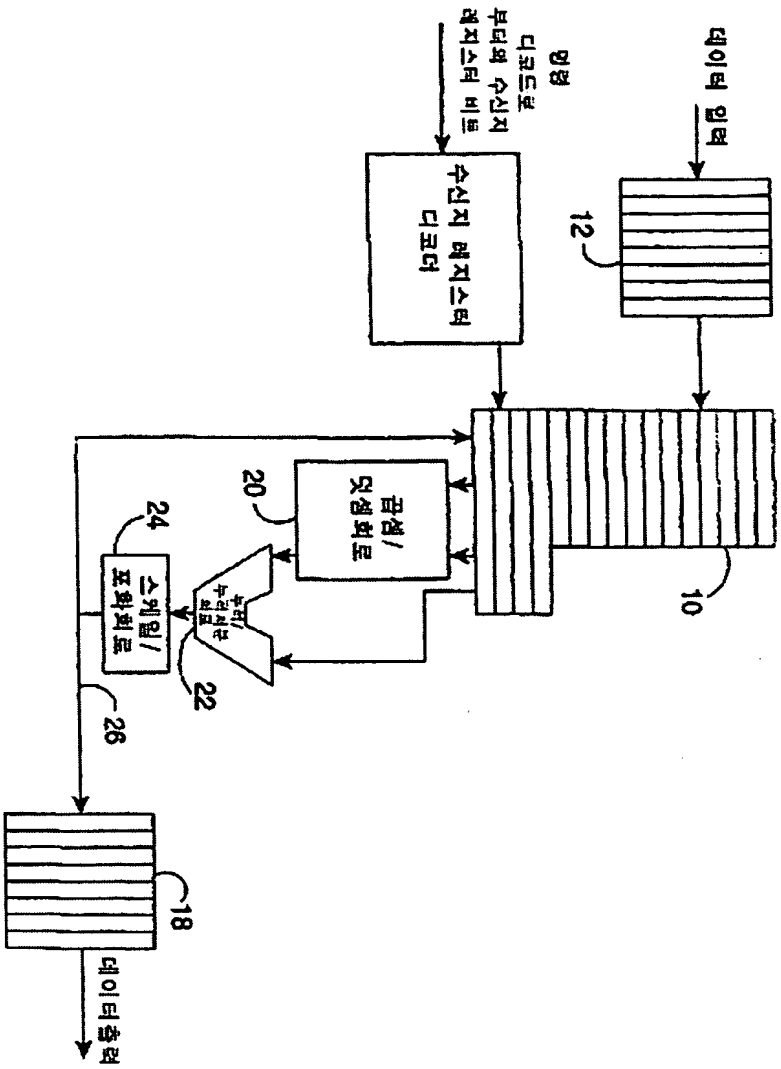
FIR (블록 필터 알고리즘)

	d0	d1	d2	d3	d4	d5
A0=	c0	c1	c2	c3	c4	c5
A1=		c0	c1	c2	c3	c4
A2=			c0	c1	c2	c3
A3=				c0	c1	c2

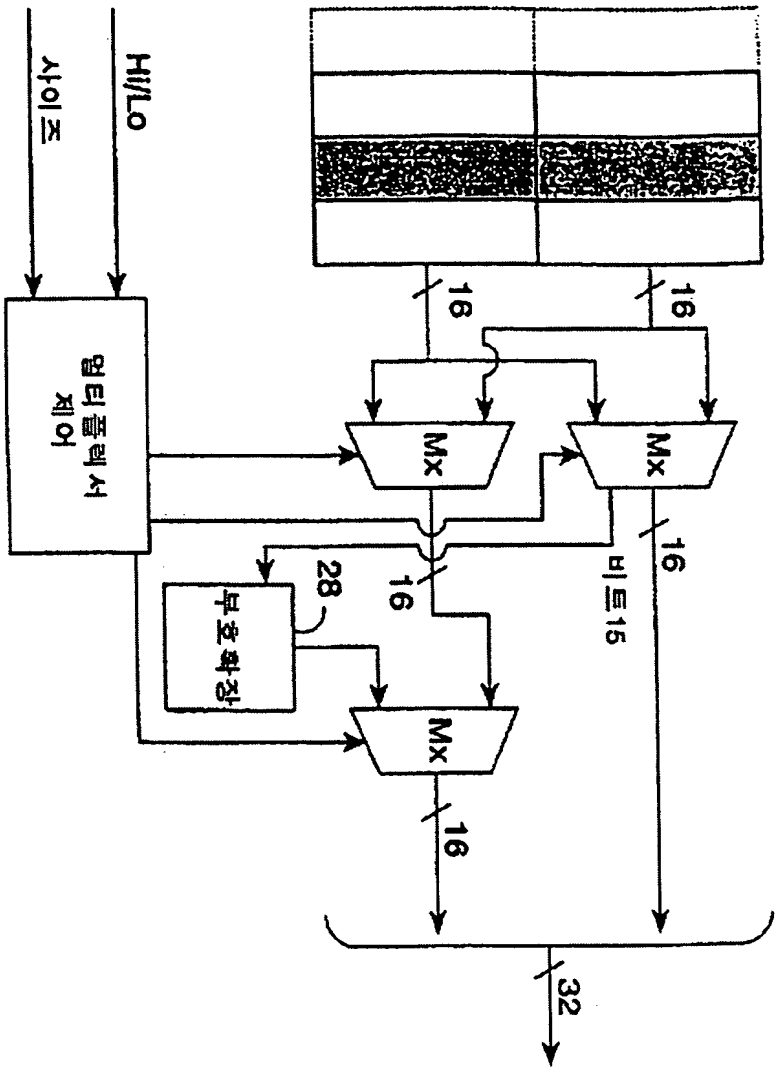
도면2


CUP에 의하여 삽입
12
저장실
H
Rn
제충전 제어
16
CP에 의하여 이동
Rn
E
16
CP 레지스터 선택
14
코프로세서 레지스터
10

도면3


데이터 입력
입력
디코드된
부대역 수신지
레지스터 비트
수신지 레지스터
디코더
12
10
20
곱셈/
덧셈회로
24
스케일/
포화회로
22
26
18
데이터 출력

도면4



도면5